DOI: https://doi.org/10.23939/csn2024.02.142
УДК 004.9, 004.62, 004.627

ОЦІНКА ЕФЕКТИВНОСТІ ТА ПРОДУКТИВНОСТІ ФОРМАТІВ СЕРІАЛІЗАЦІЇ ДЛЯ РОЗПОДІЛЕНИХ СИСТЕМ

Е.Є. Мальцев, О.В. Муляревич

Національний університет «Львівська політехніка»
кафедра електронних обчислювальних машин
E-mail: eduard.y.maltsev@lpnu.ua, oleksandr.v.muliarevych@lpnu.ua

Проведене дослідження дозволяє оцінити вплив різних форматів серіалізації на продуктивність міжсервісного комунікації, концентруючись на швидкості серіалізації, ефективності смуги передачі даних та затримці в середовищах, які інтегрують проміжне програмне забезпечення, що є характерним для мікросервісних архітектур. Через емпіричний аналіз широкого спектра форматів серіалізації та порівняння з традиційними стандартами, демонструється, що компактність серіалізованих форматів даних є більш критичною для зменшення кінцевої затримки, ніж сама швидкість серіалізації. Незважаючи на високу швидкість серіалізації, такі протоколи як FlatBuffers і Cap'n Proto показують нижчу продуктивність у розподілених середовищах через більший розмір повідомлень, на відміну від більш збалансованої продуктивності, що спостерігається у протоколах Avro, Thrift і Protobuf. Мета статті – провести огляд існуючих форматів даних та протоколів обробки та передачі повідомлень, шляхом практичних експериментів продемонструвати важливість оптимізації розміру повідомлень для підвищення ефективності мережі та її пропускної здатності.

Ключові слова: кодування даних, оцінка продуктивності, протоколи передачі повідомлень, розподілена система, формати даних.

1. Вступ

У розподілених обчисленнях ефективність міжсервісної комунікації напряму впливає на показники продуктивності, масштабованості та надійності системи. Розподілені системи використовуються для підтримки дедалі більшої кількості застосувань у різних технологіях – від хмарних обчислень і мікросервісів до великих даних і Інтернету речей (IoT), це призвело до появи великої кількості різних форматів даних та протоколів обміну інформацією. У розподілених обчисленнях, коли йдеться про формат серіалізації для обміну даними, мова йде про рівні моделі OSI (Open Systems Interconnection), які залучені до передачі даних між системами. Формат серіалізації впливає на ефективність і продуктивність на певних рівнях цієї моделі. Зокрема, формат серіалізації для обміну даними переважно стосується шостого рівня (Presentation Layer) і сьомого рівня (Application Layer) моделі OSI.

Сьомий рівень моделі OSI описує взаємодію безпосередньо з прикладними програмами, які потребують обміну даними через мережу. Формати серіалізації, визначені на цьому рівні, впливають на протоколи прикладного рівня, такі як HTTP, gRPC, MQTT тощо. В рамках даної роботи акцент було зроблено на дослідження шостого рівня моделі OSI, який відповідає за форматування і кодування даних для їх подальшої передачі або зберігання. Саме тут здійснюється серіалізація та десеріалізація даних, тобто перетворення структур даних у формат, придатний для передачі через мережу

(серіалізація) і зворотний процес (десеріалізація). На цьому рівні забезпечується узгодженість даних між різними системами, включаючи обробку таких форматів, як JSON, XML, Protobuf, Avro тощо. Вибір конкретного формату серіалізації впливає на швидкість перетворення, розмір повідомлень та сумісність між сервісами.

Незважаючи на критичну роль серіалізації у розподілених системах, в цій галузі бракує глибокого порівняльного аналізу, який би охоплював широкий спектр доступних форматів, особливо з урахуванням останніх технологічних досягнень та еволюційних вимог сучасних застосувань. Розробники та архітектори систем часто стикаються з цим складним вибором, балансуючи між швидкістю, розміром, сумісністю та зручністю використання без чітких емпіричних орієнтирів.

Дане дослідження пропонує систематичну оцінку форматів серіалізації в контексті міжсервісної комунікації в розподілених системах. Аналізуючи вибір широко використовуваних та нових форматів, це дослідження спрямоване на висвітлення характеристик продуктивності, компромісів ефективності та практичних аспектів, які визначають оптимальний вибір технології серіалізації. Зокрема, наводячи запропоновану класифікацію існуючих форматів за їх бінарною або текстовою природою, вимогами до схем та додатковими функціями, такими як можливості zero-copy, що відповідає специфічним вимогам різних системних архітектур.

2. Огляд літературних джерел

Проведемо огляд останні дослідження, щоб зрозуміти поточний стан форматів серіалізації та їх вплив на міжсервісну комунікацію в розподілених системах. Дослідження [1] порівнює формати JSON/XML з Protobuf для серіалізації даних у веб-сервісах, підкреслюючи ефективність, читабельність та забезпечення схем. JSON/XML популярні в REST через їх текстові формати, що легко сприймаються людиною, дозволяючи динамічний, безсхемний обмін даними. У статті також зазначається, що Protobuf перевершує JSON/XML в контексті gRpc за швидкістю та ефективністю використання пропускної здатності через його бінарну природу, в той час як JSON є кращим в читабельності та зручності для розробників, що сприяє налагодженню та інтеграції.

Інше цікаве дослідження [2] зосереджується на оптимізації міжсервісної комунікації в хмарному середовищі мікросервісної архітектури. У дослідженні представлені Protocol Buffers як незалежний за мовою програмування та апаратною платформою формат для серіалізації структурованих даних, відомий своєю ефективністю та перевагами у продуктивності над традиційними форматами серіалізації, такими як XML або JSON. Критерії оцінки включають зниження затримок, використання мережевих ресурсів та загальне покращення часу відгуку додатків у розподіленій мікросервісній архітектурі на базі Google Kubernetes, з використанням Apache JMeter для генерації навантаження та тестування продуктивності.

Дослідження [3] порівнює різні формати серіалізації, зосереджуючись на комунікації транспортних засобів з хмарою. Це актуально для нашого дослідження, оскільки формати серіалізації є фундаментальними для ефективного обміну даними в таких архітектурах. У статті оцінюються Protobuf та Flatbuffers, два бінарні формати серіалізації. Основні висновки показують, що Protobuf забезпечує швидшу серіалізацію та менший розмір повідомлень, тоді як Flatbuffers забезпечує краще використання пам'яті та швидший час десеріалізації.

Ще одне джерело [4] досліджує бінарні та текстові формати серіалізації для міжсервісної комунікації в середовищі Java мікросервісів під управлінням K-Native на Kubernetes, що узгоджується з метою нашого дослідження щодо оцінки форматів серіалізації в розподілених системах. У ньому порівнюються Protocol Buffers (бінарний формат) та JSON (текстовий формат), зосереджуючись на продуктивності та ефективності в хмарному середовищі. Формати аналізуються з точки зору їх впливу на метрики продуктивності в Java мікросервісах. Дослідження показує, що Protocol Buffers значно покращують продуктивність часу відгуку та розміру даних, особливо з комплексними та великими структурами даних. Однак, переваги Protobuf над JSON більш виражені в середовищах, де критичними є час відгуку та ефективність мережі.

Аналогічно, [5] оцінює різні протоколи серіалізації для покращення ефективності міжсервісної комунікації в dCache, розподіленій системі зберігання даних. Вирішується потреба замінити Java

Object Serialization для підвищення швидкості передачі повідомлень та зменшення часу обробки запитів. Оцінювані формати серіалізації включають Apache Avro, Fast-Serialization (FST), Java Object Serialization (JOS), Kryo та Protocol Buffers (Protobuf). Основні критерії оцінки – продуктивність (швидкість серіалізації та розмір повідомлень), підтримка еволюції схем, зручність використання, документація. Для проведення тестування продуктивності кожного протоколу серіалізації та десеріалізації використовувалась Java Microbenchmark Harness (JMH). Protobuf був визначений як найшвидший серіалізатор з найкращою підтримкою еволюції схем, FST серіалізатор був визначений для миттєвих оновлень через легкість інтеграції, що призвело до зменшення часу обробки ізольованих повідомлень на 10% у тестових екземплярах.

Розширюючи цю тему, [6] оцінює різні формати серіалізації даних, що може допомогти приймати рішення щодо найбільш ефективних методів обміну даними в розподілених системах. Оцінювані формати серіалізації включають JSON, MessagePack, NanoPB (Protocol Buffers) та XDR (eXternal Data Representation). Методології включають вимірювання тактових циклів для серіалізації/десеріалізації, використання пам'яті та розмір даних на різних мікроконтролерах (8-бітний ATmega328P, 16-бітний MSP430FR5994, 32-бітний ESP8266 та 32-бітний STM32F446RE). Основні висновки показують, що NanoPB та XDR є найбільш енергоефективними форматами. NanoPB краще підходить для мікроконтролерів з низьким енергоспоживанням і повільними радіомодулями, тоді як XDR більш ефективний для пристроїв з швидшими радіомодулями.

Джерело [7] оцінює ефективність і продуктивність форматів серіалізації в розподілених системах, зосереджуючись на мережах датчиків IoT. Оцінювані формати включають JSON, BSON, Protocol Buffers, XML, YAML, MessagePack, Apache Thrift, Apache Avro і PSON. Оцінювання проводилось на апаратних тестах Arduino UNO та ESP32-WROVER-B, використовуючи різні бібліотеки та навантаження для симуляції реальних сценаріїв IoT. PSON виявився одним із найшвидших форматів для завершення тестів кодування та декодування на ESP32, за ним слідували Protocol Buffers і MessagePack. Protocol Buffers або Apache Thrift були найефективнішими засобами кодування інформації на основі наданої інформації. Однак вони підходять лише для випадків, коли мікроконтролер і сервер знають структуру повідомлення заздалегідь, що не є типовим сценарієм в екосистемі IoT.

Джерело [8] оцінює різні JSON-сумісні бінарні специфікації серіалізації, зосереджуючись на JSON-сумісних форматах, стаття узгоджується із сучасними веб-сервісами та додатками, що робить її висновки дуже актуальними для підвищення ефективності та продуктивності міжсервісної комунікації. У дослідженні оцінюються кілька схемоорієнтованих та безсхемних бінарних специфікацій серіалізації, сумісних з JSON, включаючи, але не обмежуючись ASN.1, Apache Avro, Microsoft Bond, Cap’n Proto, FlatBuffers та іншими. Дослідження виявило значні відмінності в ефективності використання простору та зручності використання між різними специфікаціями.

Доповнюючи ці висновки, [9] оцінює вплив на продуктивність різних протоколів комунікації (REST, gRPC та Thrift) у мікросервісах, зосереджуючись на використанні мережі, CPU та пам'яті, а також часу відгуку. Оскільки ці протоколи використовують різні формати серіалізації (JSON для REST, Protocol Buffers для gRPC та бінарний формат для Thrift), дослідження оцінює ефективність цих методів серіалізації. Дослідники провели експерименти в контрольованому середовищі, де клієнтські та серверні додатки розміщувалися на тому самому хості, щоб виключити затримки, спричинені мережею. Thrift та gRPC перевершили REST за часом відгуку та ефективністю використання системних ресурсів, що обумовлено їхніми компактними бінарними форматами серіалізації та ефективними конструкціями протоколів.

Дослідження [10] вивчає ефективність JSONBinPack, особливо в схемоорієнтованому режимі, надає порівняльну основу, яка може покращити ефективність розподілених систем, зменшуючи розміри переданих даних та покращуючи швидкість серіалізації/десеріалізації. JSONBinPack у схемоорієнтованому та безсхемному режимах, традиційний JSON, BSON, CBOR, FlatBuffers, MessagePack, Protocol Buffers, ASN.1 PER Unaligned, Apache Avro, Apache Thrift. У дослідженні робиться висновок, що JSONBinPack перевершує традиційний JSON та бінарні формати серіалізації за ефективністю використання простору.

Дослідження [11] зосереджується на оптимізації Protobuf для серіалізації даних. Оцінювались HDVM, Redis і Protobuf для серіалізації даних JSON. Protobuf значно перевершує традиційний JSON за швидкістю, ефективністю та цілісністю даних, підкреслюючи його придатність для програм з обмеженою пропускною здатністю та реального часу в розподілених системах.

У статті [12] розглядається колонковий формат Apache Arrow та його протокол Arrow Flight, безпосередньо вирішуючи проблеми серіалізації/десеріалізації даних та ефективності передачі даних у розподілених системах. У документі порівнюється продуктивність та ефективність Arrow Flight з традиційними методами серіалізації та передачі даних. Акцент зроблено на сценаріях з високою пропускною здатністю та передачею великих наборів даних, оминаючи питання накладних витрат на налаштування Arrow Flight.

Дослідження [13] детально розглядає вплив серіалізації SOAP на ефективність комунікації, зокрема у веб-сервісах, що використовують протоколи HTTP і JMS. Дослідження, проведене в [14], вивчає ефективність форматів серіалізації в розподілених системах, зосереджуючись на пристроях IoT, підкреслює важливість енергоефективної серіалізації даних. У статті представлено формат серіалізації транзакцій (TSF) і порівняння його з форматами XML і JSON.

Дослідження [15] оцінює широкий спектр бінарних форматів серіалізації, сумісних з JSON, включаючи схемоорієнтовані (ASN.1, Apache Avro, Microsoft Bond, Cap'n Proto, FlatBuffers, Protocol Buffers, Apache Thrift) та безсхемні (BSON, CBOR, FlexBuffers, MessagePack, Smile, UBJSON) формати. Критерії оцінки включають ефективність використання простору, зокрема, зосереджуючись на середньому та медіанному зменшенні розмірів серіалізованих даних у порівнянні з JSON. Схемоорієнтовані специфікації, особливо ASN.1 PER Unaligned та Apache Avro (без рамок), визначаються як найбільш ефективні з точки зору використання простору. Майбутні дослідження могли б розширитися на включення цих форматів і вивчення впливу серіалізації на інші метрики продуктивності, такі як швидкість серіалізації/десеріалізації та використання CPU/пам'яті, а також тестування в режимі реального часу для розуміння продуктивності під навантаженням та кількісної оцінки впливу різних форматів серіалізації на пропускну здатність мережі та затримку в розподілених системах. Крім того, для проведення власного дослідження важливо розуміти внутрішню роботу різних оптимізованих форматів, таких як Protocol Buffers [16], щоб краще зрозуміти сценарії, для яких вони підходять [17].

Недавнє дослідження [18] підкреслює, що вивчення альтернативних форматів веб-архівування, зокрема Parquet та Avro, показало значні покращення продуктивності у порівнянні з традиційним форматом WARC. Дослідження [19] надає цінну інформацію про те, як різні системи на основі RDMA працюють в однакових умовах. Воно підкреслює унікальні переваги підходу HatRPC для оптимізації сервісів Thrift RPC через RDMA. Висновки в [20] свідчать, що Cap'n proto швидший за Flatbuffers за часом серіалізації/десеріалізації. Джерело [21] припускає, що MessagePack (MsgPuck) перевершує інші бібліотеки з такими форматами, як Flatbuffers та NanoPB (Protobuf).

Враховуючи те що обраний формат серіалізації даних може нівелювати переваги операцій вводу/виводу через вимоги до CPU для читання, трансформації та відправлення повідомлень (що призводить до додаткових копій пам'яті), деякі науковці [22] досліджували як зробити цей процес більш ефективним. Вони пропонують використання спеціального апаратного забезпечення, такого як FPGA. Проте, це дослідження звужує свій фокус до вивчення того, як різні формати серіалізації впливають на ефективність міжсервісної комунікації, не враховуючи інші аспекти при виборі формату серіалізації.

3. Постановка задачі

Основна мета дослідження – оцінити, як різні формати серіалізації впливають на затримку міжсервісної комунікації в розподілених системах порівняно з JSON, який є традиційним форматом веб та міжсервісної комунікації в розподілених системах. Для досягнення цієї мети розіб'ємо її на менші частини. По-перше, виберемо набір кросплатформених форматів серіалізації. Потім створимо шаблонну схему даних для наших тестів і перетворимо її на відповідні схеми для кожного формату, що працює використовуючи схеми для роботи з даними. Далі, виміряємо швидкість

серіалізації/десеріалізації та показники ефективності використання дискового простору, щоб краще зрозуміти вплив складових частин на загальну метрику затримки. Потім проведемо симуляційний тест у розподіленому середовищі для вимірювання затримки для кожного обраного формату. І як підсумок – проведемо порівняльний аналіз зібраних метрик і надамо висновки по ефективності використання одних чи інших форматів на шостому рівні моделі OSI.

4. Вибір форматів серіалізації

У нашому дослідженні було вирішено оминути формати серіалізації, залежні від платформи, такі як Java serialization, .NET Binary Formatter і Python's Pickle, через їхню відсутність універсальної сумісності та інтероперабельності в різних обчислювальних середовищах. Хоча ці формати ефективні в межах своїх екосистем, вони не відповідають критерію широкого застосування. При виборі було використано як індикатори – частоту пошукових запитів у Google та GitHub, статистику завантажень бібліотек та інші дослідження з оцінки продуктивності, в той же час зосереджуючись на швидкості серіалізації/десеріалізації, ефективності зберігання та використанні пропускної здатності мережі. Такий підхід дозволив визначити теоретично перспективні формати, що мають вже широке практичне використання. Специфічні формати, такі як TSF, Apache Arrow і PSON, були свідомо виключені з аналізу через їх заявлені специфічні моделі використання. Однак було включено JsonBinPack для спеціалізованого тестування ефективності зберігання, незважаючи на його обмежену застосовність у середовищах, заснованих на JVM, щоб підкреслити його потенціал у оптимізації зберігання даних у процесах серіалізації. Обрані формати серіалізації та відповідні бібліотеки JVM та інструменти, які використовуються, наведені в Таблиці 1.

Таблиця 1

Обрані формати серіалізації

№	Формат	Версія використаної бібліотеки
1	Apache Avro	1.11.0
2	Protocol Buffers	protobuf-java, 3.22.2
3	Thrift	libthrift, 0.19
4	JsonBinPack	npm, jsonbinpack@1.1.2
5	Flatbuffers	flatbuffers-java 23.5.26
6	Cap'n (unpacked)	java, org.capnproto.runtime 0.1.16, capnp 1.0.2
7	Cap'n (packed)	java, org.capnproto.runtime 0.1.16, capnp 1.0.2
8	MessagePack	jackson-dataformat-msgpack, 0.9.8
9	BSON	bson4jackson, 2.15.0
10	CBOR	jackson-dataformat-cbor, 2.14.2
11	AmazonIon	jackson-dataformat-ion, 2.14.2
12	Smile	jackson-dataformat-smile, 2.14.2
13	JSON	jackson, 2.14.2
14	XML	jackson-dataformat-xml, 2.14.2
15	YAML	jackson-dataformat-yaml, 2.14.2

Далі наведено короткий опис деяких обраних форматів. Apache Avro – компактний, швидкий бінарний формат з багатими структурами даних та надійним, компактним та ефективним механізмом серіалізації. Він розроблений для серіалізації даних незалежно від мови програмування і часто використовується в Apache Hadoop для обробки великих даних. Protocol Buffers – розроблений компанією Google, відомий своєю простотою та ефективністю, дозволяє серіалізувати структуровані дані. Широко використовується у внутрішніх та зовнішніх сервісах Google. Thrift – спочатку розроблений Facebook, поєднує програмний стек із механізмом генерації коду для побудови сервісів, які працюють ефективно та безперешкодно між багатьма мовами програмування. JsonBinPack – ефективний бінарний формат, спрямований на мінімізацію розміру JSON-документів, зосереджений переважно на ефективності зберігання, що особливо корисно для веб та мобільних додатків, де мають

значення пропускна здатність та зберігання. FlatBuffers – розроблений компанією Google, цей формат з «нульовим копіюванням» призначений для високої продуктивності з витратами на ефективність пам'яті. Дозволяє прямий доступ до серіалізованих даних без розбору/розпаковування, що робить його ідеальним для реальних додатків у специфічних сценаріях. Cap'n Proto – акцентує увагу на швидкості, дозволяючи отримувати доступ до серіалізованих даних без розбору, як і FlatBuffers. У цьому дослідженні Cap'n Proto з запакованим кодуванням буде називатися Cap'n (packed) для стислості; відповідно, буде використовуватись Cap'n (unpacked) для кодування без запакування.

5. Підготовка тестових даних

Для проведення тестування з форматами серіалізації, що базуються на схемах, потрібна тестова схема. Необхідно також, щоб схема відображала складність реальних навантажень, з якими часто стикаються в різних бізнес-сценаріях. Ця схема повинна мати помірний до високого рівень вкладеності, щоб максимально відповідати практичним випадкам використання. Крім того, вона повинна бути структурована таким чином, щоб дозволити безперебійну та перевірену конверсію в інші формати серіалізації, такі як ProtoBuf або Thrift. Частину Java POJO представлення тестової структури показано на Рис.1. Тестова схема Avro буде відома як Базова Схема. Вона представляє складну та реалістичну структуру даних, яка може зустрічатися в реальних фінансових транзакціях; складність, включаючи вкладені записи (наприклад, AmountDetail, AccountDetail), масиви, мапи, перерахування та різні типи даних (рядки, числа з рухомою комою, масиви, мапи), забезпечує комплексну тестову базу для оцінки продуктивності та можливостей різних форматів серіалізації.

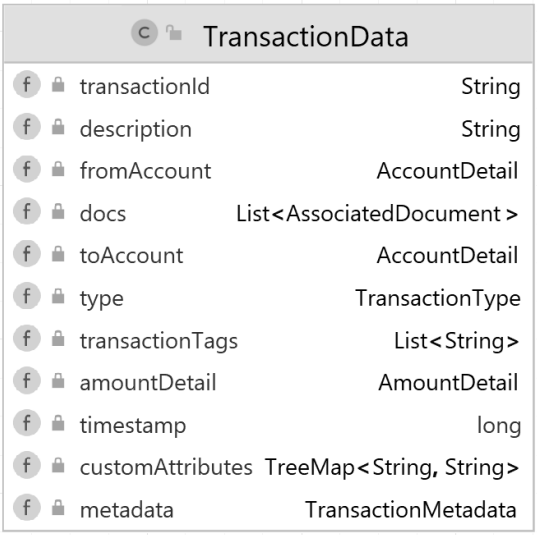

Рис. 1. Фрагмент тестової схеми Java POJO.

Для забезпечення узгодженості тестових сценаріїв було відтворено базову схему у всіх цільових форматах серіалізації. Пряма відповідність не завжди можлива; було виконано ряд трансформацій та підібрано найближчі представлення схеми цільового формату, включаючи відповідність типів, структур, та допустимості наявності null значень.

Наприклад, для Thrift довелося виконати ряд трансформацій до базової схеми, як описано нижче:

1) Пряма трансляція перерахувань Currency і TransactionType, використовуючи підтримку перерахувань у Avro та Thrift.
2) Складні записи Avro, такі як GeographicCoordinates, Address, AccountDetail і т.д., конвертуються в еквівалентні структури Thrift, зберігаючи вкладені структури даних.
3) Пряма конвертація примітивних типів Avro (наприклад, рядок, число з рухомою комою, довге число, булевий тип) на відповідні типи Thrift (рядок, double, i64, bool).
4) Масиви Avro перетворюються на списки Thrift (наприклад, list<string>), а мапи Avro - на мапи Thrift, зберігаючи семантику колекцій.
5) Тип bytes Avro для бінарних даних безпосередньо конвертується у бінарний тип Thrift.

6) Обробка nullable типів Avro через механізм опціональних полів Thrift, використовуючи присутність або відсутність поля для управління можливістю наявності null значень.
7) Призначення унікальних ідентифікаторів полів у Thrift для кожного поля структури є вимогою для серіалізації Thrift та управління еволюцією полів.
8) Перетворення типу long Avro для позначок часу на i64 Thrift.

Частину результуючої схеми Thrift показано на Рис. 2 – результат описаних дій по конвертації з базової схеми.

```
struct TransactionMetadata {
  1: list<AuthorizationDetail>
               authorizationChain,
  2: string deviceUsed,
  3: bool isRecurring,
  4: string location,
  5: string merchant
}

struct TransactionData {
  1: AmountDetail amountDetail,
  2: list<AssociatedDocument>
               associatedDocuments,
  3: map<string, string>
               customAttributes,
  4: string description,
  5: AccountDetail fromAccount,
  6: TransactionMetadata metadata,
  7: i64 timestamp,
  8: AccountDetail toAccount,
  9: string transactionId,
  10: list<string> transactionTags,
  11: TransactionType type
}
```

Рис. 2. Фрагмент тестової схеми для Thrift.

Підхід Cap’n Proto був подібний до використаного для Thrift; конвертація з Avro на Cap'n Proto включає явну нумерацію полів, трансформацію мап Avro у списки Cap'n Proto з користувацькими структурами через відсутність прямого типу мапи та присвоєння цілочисельних значень значенням Enum. Ключові відмінності включають обробку опціональних полів неявно в Cap'n Proto проти nullable полів Avro та тип bytes у Avro, що перетворюється на Data в Cap'n Proto. Для JsonBinPack підхід відповідає офіційному підходу до тестування, у якому схема генерується з JSON-файлу. Повний тестовий JSON-запис було надано як вхідний для команди jsonbinpack compile для створення схеми. На момент написання статті не існує офіційно підтримуваної бібліотеки для середовищ на базі JVM, тому замість цього було використано консольний інструмент.

Перетворення схеми Avro в ProtoBuf включає переклад типів даних та структур для збереження семантичної цілісності та узгодженості. Базові типи, такі як рядок, bytes та double, безпосередньо сумісні між Avro та ProtoBuf, що забезпечує пряме перетворення для полів, таких як amount та documentContent. Складні типи, такі як масиви та мапи, обробляються по-різному: масив Avro конвертується у повторювані поля ProtoBuf, а тип мапа Avro безпосередньо відповідає мапі ProtoBuf для пар ключ-значення. Вкладені структури зберігаються при перекладі, з вкладеними записами/повідомленнями (наприклад, AccountDetail включає Address, який включає GeographicCoordinates), що зберігає ієрархічні зв'язки даних.

Конвертація Базової Схеми у структуру FlatBuffers включає перетворення Avro Enum безпосередньо FlatBuffers Enum, забезпечуючи сумісність їх байтового представлення. Примітивні типи та масиви Avro транслюються у скалярні типи та вектори FlatBuffers відповідно, що вимагає узгодження типів. Записи Avro, що представляють складні та вкладені структури, конвертуються в

таблиці FlatBuffers, де кожен вкладений запис стає відповідною таблицею. Мапи Avro представляються як вектори пар ключ-значення в FlatBuffers, такі як StringDoublePair і StringStringPair, імітуючи структури мапи шляхом конвертації кожного запису мапи в екземпляри цих таблиць. Опціональні поля в Avro, які не мають прямої підтримки nullable у FlatBuffers, потребують керування для точного відображення відсутності даних. Тип bytes Avro для бінарних даних, як видно у document_content, безпосередньо конвертується в масив [ubyte] у FlatBuffers, забезпечуючи правильне кодування та декодування даних. Позначки часу, представлені типом long Avro, відображаються на int64 у FlatBuffers, зберігаючи точне представлення значень часу. Користувацькі атрибути, змодельовані як мапа в Avro, вимагають конвертації у вектор StringStringPair у FlatBuffers, де кожна пара ключ-значення карти Avro стає окремим екземпляром таблиці у векторі. Ця конвертація підкреслює необхідність уважного ставлення до сумісності типів, представлення структур та збереження цілісності даних у різних форматах серіалізації. Підтримка логічних типів Avro, таких як десяткові числа та дати, створює виклики при конвертації у FlatBuffers, які не мають прямих еквівалентів, що вимагає спеціальні схеми кодування або додаткові метадані для точного представлення логічних типів у FlatBuffers.

6. Проведення тестування серіалізації

Для проведення тестування було виділено три категорії форматів серіалізації, керуючись їхньою продуктивністю в завданнях серіалізації та структурованістю представлення:

1) Набір A включає формати Avro, Protobuf, Thrift, Flatbuffers, Cap’n Proto (unpacked).
2) Набір B включає формати MessagePack, BSON, CBOR, Cap’n (packed), і Smile.
3) Набір C містить формати Json, XML, YAML і AmazonIon.

Було проведено два тестових сценарія для кожного набору форматів, щоб дослідити продуктивність серіалізації з різними розмірами повідомлень. Перший тестовий сценарій, позначений як "1", включає менші повідомлення розміром від ~1 кБ до ~4 кБ. Другий тестовий сценарій, позначений як "2", оцінює продуктивність з більшими повідомленнями, що починаються від 4 кБ і збільшуються до 29 кБ в випадковому порядку. Таким чином, “Тест A1" та “Тест A2" представляють тестові сценарії форматів Набору A з меншими та більшими повідомленнями відповідно. Для довідки, розмір повідомлень у тестових сценаріях взято з серіалізованих JSON-повідомлень в одній і тій же послідовності. У цьому дослідженні використовуються терміни "Protocol Buffers" і "Protobuf" взаємозамінно, маючи на увазі один і той самий формат.

Тест A – демонструє результати тестових сценаріїв, що вимірюються в мікросекундах на операцію (µs/op) як залежність від розміру повідомлення (в байтах). Явний тренд помітний у всіх форматах – час, необхідний для операцій, загалом зростає зі збільшенням розміру повідомлення. Це інтуїтивно зрозуміло, оскільки більші повідомлення містять більше даних для серіалізації або десеріалізації. Зокрема, збільшення розміру повідомлення з приблизно 1000 до декількох тисяч байтів (з 984 до 3715) призводить до значного збільшення µs/op для всіх форматів, крім Flatbuffers, який показує відносно помірне збільшення, як показано на Рис.3 ліворуч, Тест A1.

Формат Flatbuffers постійно перевершує інші протоколи за всіма розмірами повідомлень, зберігаючи значення µs/op нижче 0,5 для всіх перевірених розмірів повідомлень у цих тестових сценаріях. Це, ймовірно, пов'язано з його ефективним підходом без копіювання, що дозволяє прямий доступ до серіалізованих даних без розпаковування. Це може зробити його придатним для додатків, де швидкість серіалізації є вузьким місцем. Цей висновок суперечить результатам, представленим у [3], де зазначено, що Protobuf має вищу швидкість серіалізації.

Хоча Protobuf не настільки продуктивний, як Flatbuffers, він демонструє чудову ефективність, особливо при менших розмірах повідомлень, а також незначне збільшення µs/op зі збільшенням розмірів повідомлень до розмірності кілобайт, що свідчить про те, що Protobuf оптимізований для повідомлень малого та середнього розміру. Avro показує помірне збільшення µs/op із більшими розмірами повідомлень, але зберігає нижчі значення µs/op порівняно з Thrift для менших повідомлень. Його продуктивність вказує на баланс між швидкістю серіалізації та гнучкістю схем, якими Avro відомий.

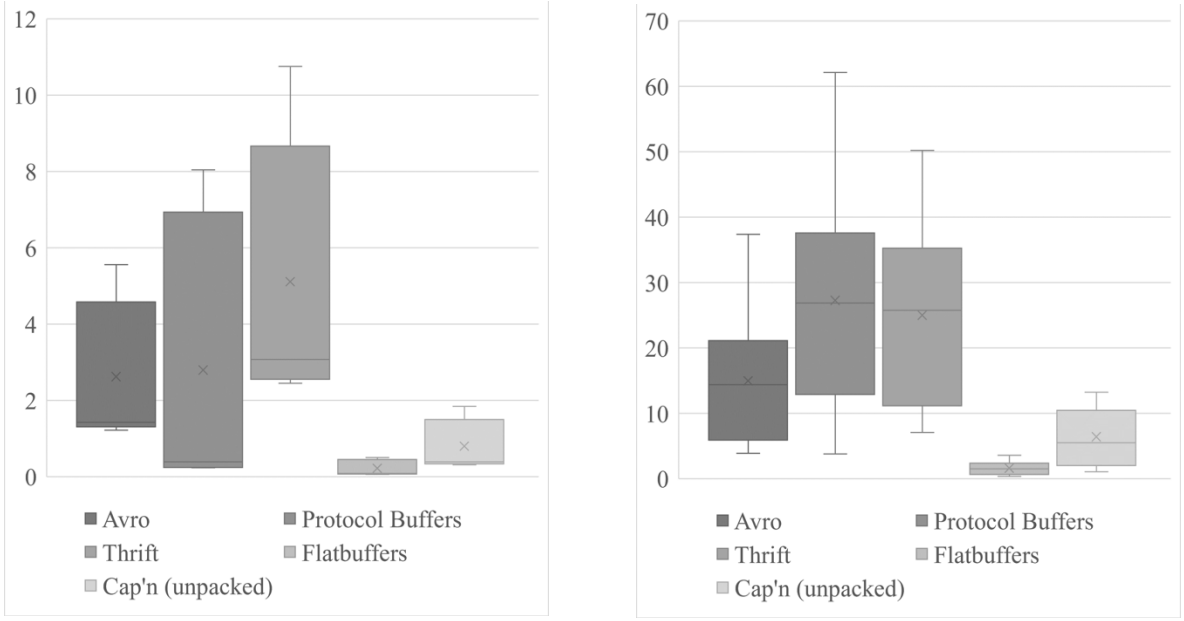

Рис. 3. Розподіл часу серіалізації (µs/op), ліворуч – Тест А1, праворуч – Тест А2.

Thrift демонструє вищі значення µs/op для всіх розмірів повідомлень у порівнянні з Avro та Protobuf. Збільшення µs/op зі збільшенням розміру повідомлень свідчить про те, що Thrift може бути менш ефективним для більших повідомлень. Однак важливо враховувати, що Thrift надає великий набір функцій для серіалізації між мовами програмування, що може виправдати компроміс у продуктивності. Проте дослідження, представлене в [9], припускає, що протокол Thrift перевершив інші через швидку серіалізацію та десеріалізацію пакетів для комунікації. Хоча в цьому дослідженні не оцінюється ефективність протоколу, можна стверджувати, що чиста швидкість серіалізації формату може не бути основним фактором продуктивності протоколу Thrift.

Cap'n Proto (unpacked) працює порівняно з Protobuf і Flatbuffers для менших повідомлень, але демонструє більше збільшення µs/op зі зростанням розмірів повідомлень у порівнянні з Flatbuffers. Це свідчить про те, що хоча Cap'n Proto розроблений для швидкості та ефективності, він може бути не придатним для більших повідомлень, в порівнянні з Flatbuffers, як показано на Рис.3 праворуч, Тест А2. Результати тестування суперечать результатам у [20], де зазначено, що Cap’n Proto має невелику перевагу у швидкості над FlatBuffers.

Ми можемо помітити, що Avro показує відносно високу продуктивність у порівнянні з Protobuf і Thrift при різних великих розмірах повідомлень. Збільшення часу серіалізації Avro є більш поступовим, ніж у Protobuf, який демонструє значне зростання часу з ростом розмірів повідомлень. Хоча час при використанні Protobuf вищий, ніж у Avro для великих повідомлень, він все ще залишається конкурентоспроможним, особливо при менших розмірах повідомлень. Thrift має тенденцію до вищих значень часу для всіх розмірів повідомлень. Формат Flatbuffers постійно демонструє найнижчі значення µs/op для всіх розмірів повідомлень, підкреслюючи його ефективність у процесах серіалізації та десеріалізації. Навіть при найбільшому розмірі повідомлення (~28 кБ) Flatbuffers зберігає час менше 5 µs/op і працює краще, ніж Protobuf, що відрізняється від результатів описаних у [3], де Protobuf показав вищі швидкості серіалізації.

Тест В – демонструє результати тестових сценаріїв з наступним набором форматів, результати якого аналогічно до Тесту А представлено на Рис.4. Як видно, формат CBOR постійно демонструє найнижчі значення µs/op для всіх розмірів повідомлень. CBOR перевершує навіть один із бінарних форматів, що базуються на схемах, а саме Cap’n (packed), у швидкості серіалізації в цьому сценарії. Наступним найкращим форматом виступає формат Smile, який пропонує конкурентні часові затрати на серіалізацію, близькі до CBOR, але має трохи більше відносне зростання з розміром повідомлення. MessagePack і BSON мають більші показники часу серіалізації, ніж CBOR і Smile, причому BSON показує найвищий час загалом, особливо для великих розмірів повідомлень. Cap'n Proto (packed)

показує середні часи серіалізації для менших повідомлень, але показує найбільший ріст часу відносно розмірності повідомлень.

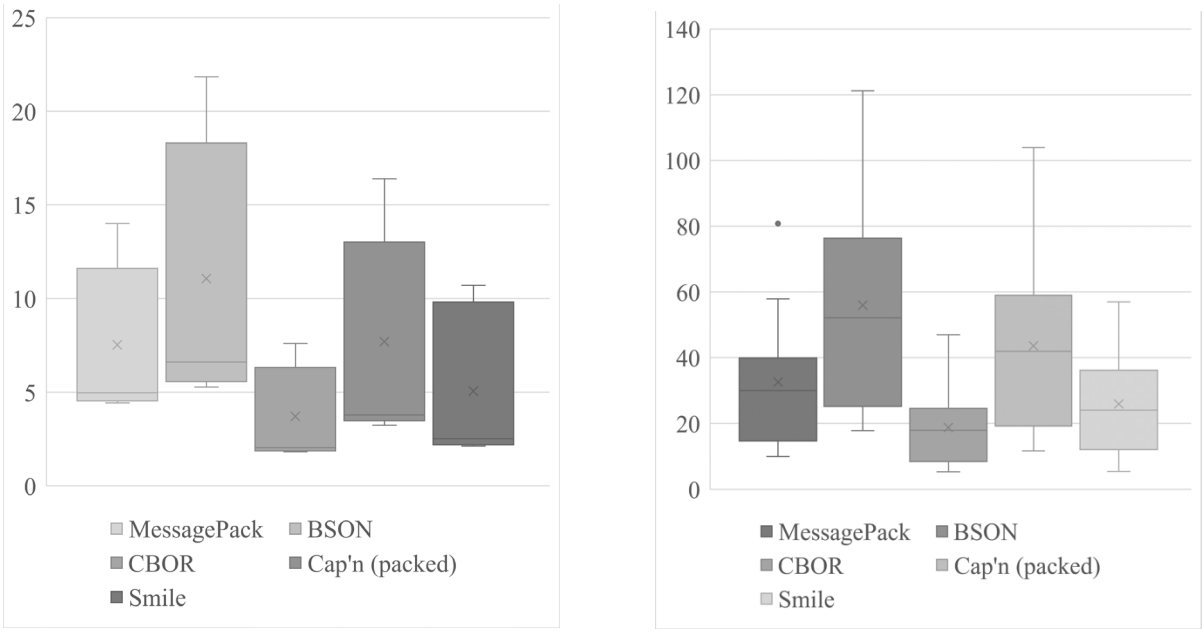

Рис. 4. Розподіл часу серіалізації (µs/op), ліворуч – Тест B1, праворуч – Тест B2.

CBOR зберігає лідерство у продуктивності з найшвидшим часом серіалізації, як показано на Рис.4. У порівнянні з MessagePack, CBOR приблизно на 18.8% швидший, демонструючи свою стійкість і масштабованість навіть при великих розмірах повідомлень (див. Рис.4 праворуч, Тест B2). Smile має час серіалізації, дуже близький до MessagePack. BSON показує найгіршу продуктивність у цих тестових сценаріях з часом серіалізації, що досягає 20 µs/op і більше. Cap'n (packed) не витримує великих повідомлень так само добре, як MessagePack і CBOR, але показує достатню продуктивність для менших повідомлень. Порівнюючи результати Тестів B1 та B2, видно тенденцію ефективності CBOR і Smile, навіть коли розмір повідомлень зростає. І навпаки, відносна неефективність BSON і Cap'n Proto стає більш вираженою при великих повідомленнях. MessagePack залишається середнім варіантом, зберігаючи постійне співвідношення продуктивності відносно інших форматів, але не перевершуючи Protobuf, на відміну від результатів, представлених у [21].

Тест C – демонструє результати тестових сценаріїв з останнім набором форматів, результати якого аналогічно до Тестів A та B представлено на Рис.5. Як видно, AmazonIon менш ефективний у операціях серіалізації, навіть у порівнянні з класичним форматом Json, з максимальними значеннями, що досягають 30 µs/op, і медіаною, значно вищою за 75 у Json. Загалом, AmazonIon показує час серіалізації вдвічі більший, ніж Json, працюючи в цьому сценарії майже ідентично серіалізації XML, як показано на Рис.5. Найгіршим по продуктивності в даному наборі виглядає формат YAML, із середнім значенням понад 65 µs/op. Його величезний час серіалізації та діапазон понад 100 µs/op у порівнянні з іншими форматами є хорошим показником того, що швидкість серіалізації не є тим, для чого був створений YAML – зручна читабельність.

Зі збільшенням розміру повідомлення (див. Рис.5 праворуч, Тест C2) AmazonIon демонструє ще гіршу продуктивність серіалізації, ніж XML, із піками, що досягають 200 µs/op. Для порівняння, XML формат має максимуми нижче 150 µs/op. Неочікувано, у цьому сценарії Json зберігає свою перевагу з медіаною нижче 50 µs/op і з мінімальним загальним діапазоном, показуючи сильнішу стійкість до збільшення розмірів повідомлень. YAML відстає від інших форматів з медіаною понад 350 µs/op і найбільшим загальним діапазоном.

Найбільша спільнота, сильна підтримка в Інтернеті та екосистема, ймовірно, сприяли перемозі Json у тестових сценаріях C1 та C2. YAML в основному є форматом конфігурації, який пріоритетом ставить людську читабельність і легкість редагування в його текстовому форматі, а не гнучкість обміну даними. XML показав пристойну продуктивність і досі широко використовується та підтримується, що

Оцінка ефективності та продуктивності форматів серіалізації для розподілених систем означає, що багато неефективності у алгоритмах серіалізації та бібліотеках були виправлені давно.

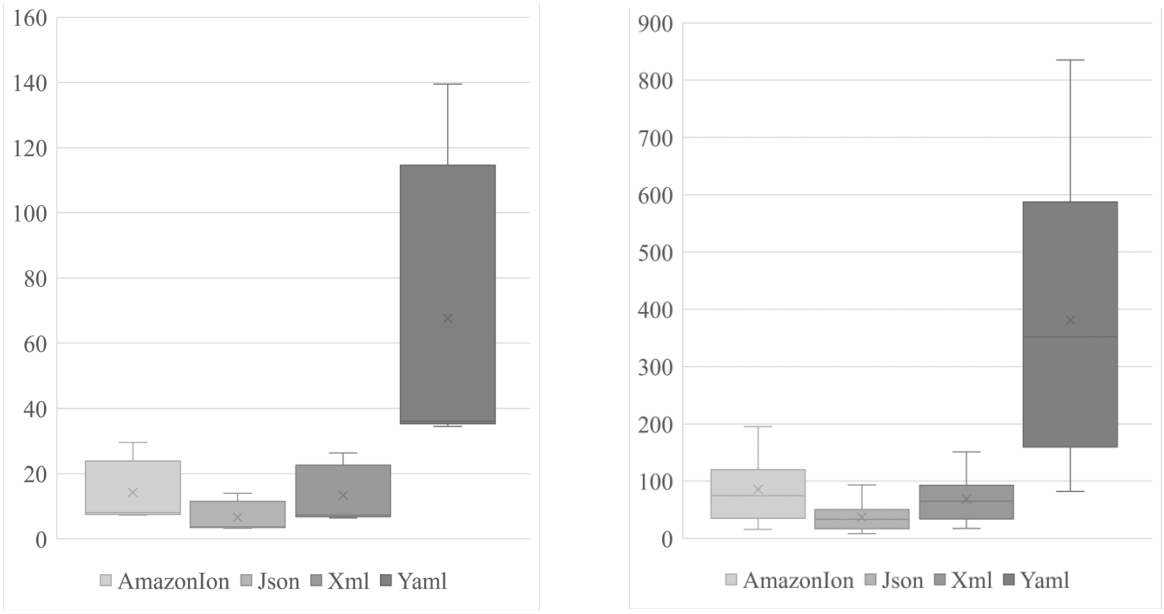

Рис. 5. Розподіл часу серіалізації (µs/op), ліворуч – Тест C1, праворуч – Тест C2.

7. Ефективність використання дискового простору

В цьому розділі наведено результати тестування ефективності використання дискового простору при застосуванні обраних форматів. Методологія для цих тестових сценаріїв подібна до вимірювань продуктивності серіалізації, але без попередньої підготовки компілятора, основним показником тут буде розмір байтового масиву, створеного кожним форматом у кожній ітерації. Налаштування ітерацій такі ж, як і раніше, з розмірами повідомлень від ~1 кБ до ~30 кБ; у цьому сценарії немає розділення тестів на дві групи, а вимірюються різні характеристики статистичного розподілу протягом 30 ітерацій одразу в псевдовипадковому порядку та генеруються дані. Зібрані метрики будуть представлені у вигляді таблиці із зазначенням середніх, медіанних та стандартних відхилень для кожного формату, впорядкованих за медіаною у зростаючому порядку.

Давайте подивимося на розраховані статистичні показники, представлені в Таблиці 2. У цьому сценарії Avro є найбільш ефективним форматом з точки зору розміру, з середнім розміром серіалізованого вихідного файлу 4871,7 байт, за ним йде JsonBinPack. Це суперечить висновкам із [10], які свідчать, що JsonBinPack перевершує Avro за просторовою ефективністю. У цьому дослідженні JsonBinPack також поступається Avro у медіанній просторовій ефективності на 2,4 % та середній на 2,63 % за всіма розмірами повідомлень і має вищий стандарт відхилення.

Далі йде Thrift, який демонструє середній розмір 5002,6 байт. Thrift показує компактний середній розмір і підтримує менший медіанний розмір, що вказує на стабільну продуктивність на різних наборах даних. Наступний найкращий результат показує Protobuf, який відстає на 4,33 % за медіаною та на 4,44 % за середнім показником просторової ефективності. Protobuf показав медіану 2571,5 байт у цьому наборі даних і має ширший діапазон, ніж Thrift.

Неочікувано, Smile показує медіанний розмір 3016 байт. Smile перевершив Cap'n (packed) на 0,13 % за медіаною та на 5,92 % за середнім розміром. Cap'n (packed), однак, перевершив MessagePack на 7 % як за медіаною, так і за середнім розміром. CBOR досить близький за показниками до MessagePack, на 0,3 % більшим середнім і на 0,45 % медіанним розміром байтів і трохи більшим діапазоном. CBOR перевершив AmazonIon на 13,9 % за середнім розміром і на 12,86 % за медіанним розміром. AmazonIon був кращим, ніж Json, на 1,6 % за середнім і на 2,4 % за медіаною в цьому тесті просторової ефективності та показав трохи тісніший діапазон.

JSON перевершив BSON на 2,4 % за середнім і на 3,35 % за медіанним розміром, а також FlatBuffers на 2,1 %, та Cap'n (unpacked) на 0,39 % за медіанним розміром відповідно.

Таблиця 2

Розмірність серіалізованих даних після 30 ітерацій

№	Формат	Середнє значення, байт	Медіана, байт	Стандартне відхилення, байт
1	Avro	4871.7	2368.5	5621.82
2	JsonBinPack	4999.9	2427	5770.67
3	Thrift	5002.6	2459.5	5707.69
4	Protobuf	5225.0	2571	5990.10
5	Smile	5685.8	3016	5993.60
6	Cap'n (packed)	6022.9	3020	6799.23
7	MessagePack	6486.0	3250.5	6760.11
8	CBOR	6511.7	3265.5	6776.64
9	AmazonIon	7418.3	3747.5	7632.56
10	JSON	7542.8	3841	7673.64
11	BSON	7728.0	3974.5	7877.43
12	Flatbuffers	7658.4	4060	8113.77
13	Cap'n (unpacked)	7681.0	4076	8176.29
14	YAML	8251.3	4226.5	8404.03
15	Xml	15040.8	8082.5	15424.14

YAML був менш ефективним у просторі, ніж Cap’n (unpacked) на 7,4 % за середнім і на 3,5 % за медіанним розміром. На іншому кінці спектру XML демонструє найменшу просторову ефективність, із середнім розміром 15040,8 байт і медіаною 8082,5 байт. Він відстає від YAML на 82 % за середнім і на 47,7 % за медіанним розміром. Значна різниця в розмірах підкреслює перевантаженість формату XML та відповідний вплив на ефективність зберігання та передачі. Слід відзначити дещо іншу картину, якщо мова йде про менші розміри повідомлень, результати досліджень представлено в Таблиці 3. Основні зміни полягають у тому, що Protobuf і Cap’n Proto демонструють значно кращі показники.

Таблиця 3

Розмірність серіалізованих даних після перших 15 ітерацій

№	Формат	Середнє значення, байт	Медіана, байт	Стандартне відхилення, байт
1	Avro	862.9	160	959.9
2	JsonBinPack	890.4	175	975.8
3	Thrift	932.6	217	975.7
4	Protobuf	945.8	186	1038.8
5	Smile	1403.0	631	1048.0
6	Cap'n (packed)	1173.6	307	1178.0
7	MessagePack	1655.2	744	1225.8
8	CBOR	1670	756	1229.4
9	AmazonIon	1951.2	898	1409.7
10	JSON	2048.0	984	1423.4
11	BSON	2073.4	954	1490.4
12	Flatbuffers	1803.7	632	1563.3
13	Cap'n (unpacked)	1801.0	640	1551.7
14	YAML	2223.5	1038	1578.5
15	Xml	3788.7	1297	3272.6

Це може свідчити про те, що Protobuf є більш ефективним, ніж Thrift для менших розмірів повідомлень. Також Cap’n (packed) перевершує Smile на 19 % для менших повідомлень у цьому

випадку, і це може сприяти загальному тесту розподіленої затримки, оскільки цей тест переважно працює з подібним діапазоном розмірів повідомлень. Flatbuffers також піднявся на вищу позиції в таблиці в цьому сценарії, що може бути пов'язано з тим, як формати без копіювання (zero-copy) обробляють внутрішні буфери байтів і їх поступове збільшення з ростом розмірності повідомлення. Аналогічно і для Cap'n Proto (unpacked), оскільки він використовує подібні техніки та методи, як і формат Flatbuffers.

8. Тестування затримки в розподіленому середовищі

Для цього тестового сценарію буде використано три потоки Java Virtual Machine (JVM), які діють як окремі компоненти розподіленої системи; з відповідними назвами – Сервіс A, B і C. Ці потоки розроблені для імітації ролей «виробників» (producers) і «споживачів» (consumers) у розподіленій архітектурі, використовуючи Kafka як посередник для передачі повідомлень. Тестовий контейнер Kafka буде основою для цього налаштування, забезпечуючи ізольоване розгорнуте середовище, яке імітує реальні розподілені системи обміну повідомленнями. Кількість і розмір повідомлень будуть збільшуватися з кожною ітерацією у випадковому порядку до 4 кБ з загальною кількістю 15 ітерацій. Розмір тестового набору в тесті варіюється від 12000 до 30000 повідомлень. Процедура тестування охоплює такі елементи:

1) Спочатку генерується тестовий набір. Цей тестовий набір перетворюється у відповідне внутрішнє представлення для кожного формату та називається «Тестовим Набором».
2) Перед початком тестового сценарію (бенчмарку) мережеве посередництво, включаючи тестовий контейнер Kafka та відповідних споживачів і виробників, "попередньо нагрівається" (проходить етап попереднього опрацювання випадкових даних), щоб забезпечити оптимальний рівень продуктивності під час тесту.
3) Тестовий сценарій починається зі старту вимірювання часу для відстеження тривалості симуляції.
4) Сервіс A бере на себе відповідальність за серіалізацію всього набору записів і передачу його до Middleware topic B, ініціюючи потік даних у системі.
5) Після отримання «Тестового Набору» з topic B, Сервіс B обробляє його, застосовуючи попередньо визначені критерії на основі двох специфічних властивостей. Після фільтрації Сервіс B передає відфільтрований набір до Middleware topic C для подальшої обробки.
6) Сервіс C споживає відфільтрований набір з topic C, відзначаючи кінець ітерації. Цей крок включає перевірку отримання всього набору, щоб забезпечити повноту передачі даних.

Тестовий сценарій проводиться для кожного формату даних окремо, з результатами, виміряними в мілісекундах. Кожна ітерація оцінюється окремо і послідовно відносно інших. Тепер розглянемо отримані емпіричні результати представлені на Рис.6 після оцінки часових затримок при використанні цільових форматів у симуляційному середовищі розподіленої системи.

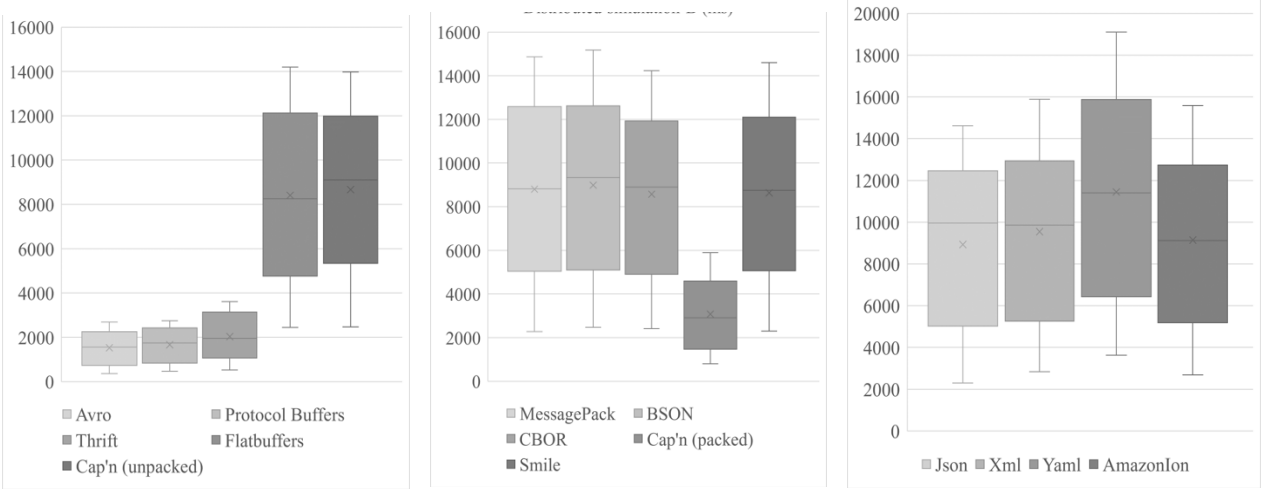

Рис. 6. Розподіл часу затримки у симуляційному середовищі (ms), зліва на право продемонстровано результати Тестів A, B та C відповідно.

Формат Protobuf має затримку більшу, ніж Avro, але меншу, ніж Thrift, як показано на Рис.6, Тест А (крайня ліва частина). Середня затримка близька до медіани, що вказує на симетричний розподіл точок даних. Формат Thrift показує найвищі медіанні та середні затримки серед трьох форматів, що вказує на найповільнішу продуктивність у цій симуляції. Thrift також має найбільший діапазон, що свідчить про більшу варіативність затримок, ніж у форматів Avro і Protobuf. Це може свідчити про те, що продуктивність формату Thrift менш передбачувана або більш чутлива до змін в умовах симуляції. Слід відзначити, що відомі виконавці у тестових сценаріях по швидкості серіалізації, такі як Flatbuffers і Cap'n (unpacked), не показали аналогічних позитивних результатів у розподіленій симуляції, як формати Avro, Protobuf і Thrift. Дані свідчать про те, що розподілена затримка у цьому сценарії визначається більше ефективністю простору, ніж швидкістю серіалізації.

Формати MessagePack, CBOR і Smile працюють подібно в цьому тесті, з медіанним значенням близько 9 секунд, як показано на Рис.6, Тест В (частина посередині). Також можна спостерігати вищу кореляцію з отриманими результатами просторової ефективності, ніж зі швидкістю серіалізації, особливо для менших розмірів повідомлень. Формат Cap'n (packed) показав відносно хороші результати в цьому тесті, з медіаною близько 3-х секунд, незважаючи на посередні результати у швидкості серіалізації, він показав набагато кращі результати у просторовій ефективності для менших розмірів повідомлень, як показано в Таблиці 3, що є головним чинником загальної затримки. Формат Protobuf також демонструє трохи кращі результати, ніж формат Thrift, що знову корелює з результатами, представленими в Таблиці 3. Незважаючи на те, що формат BSON показав найгірші результати у тесті по часу серіалізації, він продемонстрував результати з медіаною, вищою, але близькою до форматів MessagePack і CBOR.

Формат YAML є найгіршим за показниками у Тесті С з медіанним і середнім часом понад 10 секунд, як показано на Рис.6 (крайня права частина). Формат AmazonIon був найкращим виконавцем за медіанними і середніми значеннями трохи більше 9 секунд, ймовірно, через менший розмір даних при серіалізаціїї згідно результатів у Таблиці 3, якщо порівнювати з форматами Json, XML і YAML.

Застосування формату Avro дозволило отримати найкращі показники у цій розподіленій симуляції, за ним слідують формати Protobuf і Thrift. Лідери за критерієм швидкості серіалізації формати Flatbuffers і Cap'n (unpacked) не змогли наздогнати їх, ймовірно, через більший серіалізований розмір даних, що знижує ефективність мережі. Немає чіткого переможця серед бінарних форматів без схем, але варто зазначити, що Json показав хороші результати у порівнянні з бінарними форматами без схем, у багатьох випадках перевершуючи формати MessagePack і BSON. Формати CBOR і Smile показали кращі результати, ніж JSON у розподіленій симуляції, що корелює з результатами тесту по використанню простору, та відповідно розмірністю серіалізованих даних.

Використану при тестуванні симуляцію можна покращити, включивши додаткові тестові сценарії, варіації складності повідомлень і вкладеності, варіації розмірів тестового набору та розглянути додаткові формати серіалізації, такі як Microsoft Bond, PSON, UBJSON, Flexbuffers і ASN.1.

9. Результати дослідження

Бінарні формати серіалізації, що використовують схеми, такі як Avro, Protobuf і Thrift, показали найкращу продуктивність у розподілених середовищах, значно зменшуючи медіанну затримку в сценаріях розподіленої комунікації порівняно з JSON, традиційним форматом серіалізації. Використання формату Avro зменшує медіанну затримку на 630 %, формату Protobuf – на 560 %, а формату Thrift – на 510 %. Формат Cap'n (packed) дозволив зменшити медіанну затримку на 340 %. Застосування формату Flatbuffers зменшило медіанну затримку на 20 %, а формати MessagePack, CBOR і Smile – на 10-12 %. Застосування решти форматів або не зменшило затримку значною мірою (тобто показники були нижче 10 %), або навіть збільшило її порівняно з JSON. Наприклад, застосування формату YAML збільшило медіанну затримку на 13 %. Слід зазначити, що не було значної різниці в медіанній затримці при порівнянні JSON з XML, AmazonIon, BSON або Cap'n (unpacked).

Тестування проведені в розподіленому середовищі показали наступні результати:

1) Компактність повідомлень була більш значущою умовою зменшення розподіленої затримки,

ніж швидкість десеріалізації/серіалізації.

2) Найкращі бінарні формати без використання схем менш ефективні, ніж бінарні формати із використанням схем, у зменшенні розподіленої затримки.
3) Формати з нульовим копіюванням (zero-copy), незважаючи на їх надзвичайно швидку серіалізацію/десеріалізацію, не є конкурентноздатними у порівнянні з Avro, Protobuf і Thrift в умовах розподіленого середовища.

10. Висновки

В наслідок вивчення різних підходів до обробки даних на шостому рівні моделі OSI та форматів даних, можна зробити наступні висновки:

1) Формат Avro показав найкращу продуктивність у зменшенні медіанної затримки у розподілених середовищах, зменшивши її на 630 % у порівнянні з JSON.
2) Формат Protobuf дозволяє зменшити медіанну затримку на 560 %, що свідчить про його ефективність у розподілених системах.
3) Формат Thrift зменшив медіанну затримку на 510 %, що робить його одним із найбільш ефективних форматів серіалізації для розподілених систем.
4) Формат Cap'n Proto (packed) зменшив медіанну затримку на 340 %, показуючи хорошу продуктивність у розподілених сценаріях.
5) Формат Flatbuffers зменшив медіанну затримку лише на 20 %, що свідчить про його обмеження у розподілених середовищах.
6) Формат YAML є неефективним для використання у розподілених системах.
7) Відсутність суттєвої різниці в медіанній затримці між JSON, XML, AmazonIon, BSON або Cap'n Proto (unpacked) вказує на їх подібну продуктивність.
8) Компактність повідомлень є більш значущим фактором для зменшення розподіленої затримки, ніж швидкість десеріалізації/серіалізації.
9) Бінарні формати із використанням схем, такі як Avro, Protobuf та Thrift, були значно більш ефективними у зменшенні затримки в розподіленому середовищі ніж формати без схем.
10) Формати з нульовим копіюванням (zero-copy), такі як Flatbuffers і Cap'n Proto (unpacked), не змогли конкурувати з Avro, Protobuf та Thrift у розподіленому середовищі через більшу розмірність споживання простору, що знижує мережеву ефективність.

Для ефективної роботи розподілених систем необхідно ретельно підбирати формати серіалізації, орієнтуючись на їхню продуктивність та ефективність використання дискового простору.

Проведені симуляційні дослідження можна покращити, включивши додаткові тестові сценарії, варіації складності повідомлень і вкладеності, варіації розмірів тестового набору та формати серіалізації, такі як Microsoft Bond, PSON, UBJSON, Flexbuffers і ASN.1.

Список літератури

1. B. Marii, and I. Zholubak, "Features of Development and Analysis of REST Systems," ACPS, vol. 7, no. 2, pp. 121–129, Dec. 2022, DOI: 10.23939/acps2022.02.121.

2. S. Weerasinghe and I. Perera, "Optimized Strategy in Cloud-Native Environment for Inter-Service Communication in Microservices," Int. J. Onl. Eng., vol. 20, no. 01, pp. 40–57, Jan. 2024, DOI: 10.3991/ijoe.v20i01.44021.

3. D. P. Proos and N. Carlsson, "Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV," 2020 IFIP Networking Conference (Networking), Paris, France, 2020, pp. 10-18.

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

5. L. Morschel et al., "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, 2020, DOI: 10.1051/epjconf/202024505017.

6. D. Friesel and O. Spinczyk, "Data Serialization Formats for the Internet of Things," Electronic Communications of the EASST, p. Volume 80: Conference on Networked Systems 2021 (NetSys 2021), Sep. 2021, DOI: 10.14279/TUJ.ECEASST.80.1134.

7. Á. Luis, P. Casares, J. J. Cuadrado-Gallego, and M. A. Patricio, "PSON: A Serialization Format for IoT Sensor Networks," Sensors, vol. 21, no. 13, p. 4559, Jul. 2021, DOI: 10.3390/s21134559.

8. J. C. Viotti and M. Kinderkhedia, "A Survey of JSON-compatible Binary Serialization Specifications." arXiv, Jan. 10, 2022. DOI: 10.48550/arXiv.2201.02089.

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

10. J. C. Viotti and M. Kinderkhedia, "Benchmarking JSON BinPack," 2022, DOI: 10.48550/ARXIV.2211.12799.

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

12. T. Ahmad, Z. A. Ars, and H. P. Hofstee, "Benchmarking Apache Arrow Flight -- A wire-speed protocol for data transfer, querying and microservices." arXiv, Apr. 08, 2022. DOI: 10.48550/arXiv.2204.03032.

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

14. D. Evans, "Energy-Efficient Transaction Serialization for IoT Devices," j. of Comput. sci. res., vol. 2, no. 2, pp. 1–16, May 2020, DOI: 10.30564/jcsr.v2i2.1620.

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

16. Protocol Buffers Version 3 Language Specification. Accessed: Feb. 20, 2024. [Online]. Available: https://protobuf.dev/reference/protobuf/proto3-spec/

17. C. Currier, "Protocol Buffers," in Mobile Forensics – The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices, C. Hummert and D. Pawlaszczyk, Eds., Cham: Springer International Publishing, 2022, pp. 223–260. DOI: 10.1007/978-3-030-98467-0_9.

18. X. Wang and Z. Xie, "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, Aug. 2020, pp. 177–186. DOI: 10.1145/3383583.3398542.

19. T. Li, H. Shi, and X. Lu, "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, Nov. 2021, pp. 1–14. DOI: 10.1145/3458817.3476191.

20. Sorokin, K., "Benchmark comparing various data serialization libraries," [Online]. Available: https://github.com/thekvs/cpp-serializers. [Accessed: March 1, 2024].

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

22. J. Peltenburg, Á. Hadnagy, M. Brobbel, R. Morrow, and Z. Al-Ars, "Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators," in 2021 International Conference on Field-Programmable Technology (ICFPT), Dec. 2021, pp. 1–9. DOI: 10.1109/ICFPT52863.2021.9609833.

Оцінка ефективності та продуктивності форматів серіалізації для розподілених систем

EVALUATION OF EFFICIENCY AND PERFORMANCE OF SERIALIZATION FORMATS FOR DISTRIBUTED SYSTEMS

E.E. Maltsev, O.V. Muliarevych

Lviv Polytechnic National University,
Department of Computer Engineering
E-mail: : eduard.y.maltsev@lpnu.ua, oleksandr.v.muliarevych@lpnu.ua

The conducted study allows us to evaluate the impact of various serialization formats on the performance of inter-service communication, focusing on serialization speed, data bandwidth efficiency, and latency in environments integrating middleware, characteristic of microservice architectures. Through an empirical analysis of a wide range of serialization formats and comparisons with traditional standards, it is demonstrated that the compactness of serialized data formats is more critical for reducing end-to-end latency than serialization speed itself. Despite high serialization speed, protocols such as FlatBuffers and Cap'n Proto show lower performance in distributed environments due to larger message sizes, in contrast to the more balanced performance observed in protocols like Avro, Thrift, and Protobuf. The purpose of the article is to review existing data formats and message processing and transmission protocols, and through practical experiments, demonstrate the importance of optimizing message sizes to enhance network efficiency and bandwidth capacity.

Keywords: data encoding, performance evaluation, message transmission protocols, distributed system, data formats.