МАТЕМАТИЧНІ ТА ІНФОРМАЦІЙНІ МОДЕЛІ В ЕКОНОМІЦІ

УДК 004.7

Р.Б. АНДРУЩЕНКО, С.В. ЗАЙЦЕВ, А.Ю. СОЛДАТОВ

АНАЛІЗ МЕТОДІВ СЕРІАЛІЗАЦІЇ СТРУКТУРОВАНИХ ДАНИХ ДЛЯ ПЕРЕДАЧІ В ПРОТОКОЛАХ ПРИКЛАДНОГО РІВНЯ МОДЕЛІ OSI

Анотація. *Мережеве програмне забезпечення є невід'ємною частиною майже будь-якої інформаційної системи – від звичайних веб-сайтів до великих ентерпрайз-систем банків, фінансових організацій, розподілених обчислювальних систем та кластерів і датацентрів. Для того щоб структурні одиниці інформаційної системи могли спілкуватись між собою та розуміти одна одну, необхідно заздалегідь визначити «мову» (тобто, «протокол»), за допомогою якої вони будуть спілкуватись. Більшість сучасного програмного забезпечення використовує структуровані текстові або бінарні формати даних та протоколи, побудовані на базі UPD, TCP та HTTP. У даній роботі проведено аналіз існуючих форматів серіалізації структурованих даних в комп'ютерних мережах, розглянута їх внутрішня будова, виявлено їх особливості та запропоновано можливі способи вдосконалення.*

Ключові слова: *HTTP, комп'ютерна мережа, серіалізація, мережеве програмне забезпечення.*

Вступ

Аналіз тенденцій розвитку мережі Інтернет показує, що за останні десятиріччя кількість трафіку зростає неймовірними темпами [1, 2]. Наприклад, спеціалісти Cisco прогнозують, що глобальний інтернет-трафік досягне позначки 3.3 зетабайт в 2021 році [3]. Активно розвивається мережа мініатюрних пристроїв «Інтернету речей», яким також необхідно передавати дані стосовно свого стану та роботи [4]. Банківські установи, фінансові біржі та інші розподілені системи з високим навантаженням використовують у своїй роботі структуровані дані, що передаються у форматах XML, JSON, BJSON, SOAP та ін. NoSQL СУБД зазвичай мають вбудовану підтримку JSON, BJSON [5, 6]. Тенденції розвиваються таким чином, що навіть реляційні бази даних починають підтримувати ці популярні формати (наприклад, SQLite – JSON1 Extension, MySQL – Json Data Type) [7, 8].

1. Постановка проблеми

Для передачі даних в більшості випадків використовують протоколи на базі UDP та TCP. Протоколи на базі UDP корисні для передачі даних, де втрата деякої частини пакетів не є критичною. Наприклад, це можуть бути аудіо- та відеопотоки. Протоколи на базі TCP займають іншу нішу і використовуються для передачі даних, де втрата пакетів неприпустима. Зокрема, саме на базі TCP побудовані такі протоколи мережі Інтернет, як HTTP, SMTP, SSH та ін. [9, 10].

Проблема постає у тому, що текстові формати даних, такі як XML, JSON, SOAP і навіть їх бінарні аналоги (наприклад, BSON, MessagePack, UBJSON), містять надлишкову інформацію, частина якої взагалі може навіть не нести ніякого інформаційного навантаження. Це знижує ймовірність успішної передачі даних та впливає на швидкість передачі даних в мережі, особливо при використанні бездротового зв'язку, який останнім часом набуває все більшого поширення.

2. Аналіз останніх досліджень і публікацій

Кожен формат даних має документ-специфікацію, у якому описано, як потрібно кодувати/декодувати інформацію. Зокрема, проаналізовані формати:
1) текстові – JSON, XML [11, 12].
2) бінарні з серіалізацією схеми даних – BSON, Smile, MessagePack [13–15].
3) бінарні без серіалізації схеми даних – Protocol Buffers, Flat Buffers [16–17].

Аналіз показує, що формати даних можна розділити на текстові (напр., JSON) та бінарні (напр., MessagePack). Також існують формати зі строгою схемою даних (Protocol Buffers) та нестрогою схемою даних (XML, JSON). Формати зі строгою схемою більш ефективні, оскільки в даному випадку саму схему можна виключити з пакетів, що передаються, у разі якщо схема відома обом сторонам каналу передачі даних.

Protocol Buffers – один з найефективніших форматів. Є дослідження, що він може використовуватись в пристроях Інтернету речей [18].

Однак, як буде показано далі у статті, для Protocol Buffers та його аналогів необхідні додаткові ресурси для зберігання класів серіалізації/десеріалізації.

У статті [19] розглядається протокол HTTP/2. Досліджується його вплив на функціонування веб-сайтів і показано, що, на відміну від HTTP/1, він є бінарним та відповідно більш ефективним.

У статтях [20–21] розглядаються методи серіалізації даних, зокрема і текстові JSON та XML. Доведено, що бінарні аналоги не тільки займають менше місця, а й працюють швидше.

Тим не менш, у статті [22] вказується на те, що в деяких випадках як виняток використовуються текстові формати навіть для передачі бінарних даних.

Підтвердженням ефективності бінарних форматів даних у мережевих протоколах є їх використання, наприклад, в розподілених системах з числовим програмним управлінням [23] та в клієнт-серверних системах з високим навантаженням [24].

3. Мета статті

Головною метою даної роботи є визначення, аналіз та порівняння основних форматів серіалізації/десеріалізації структурованих даних, їх особливостей та внутрішньої структури, виявлення можливих шляхів їх вдосконалення.

4. Виклад основного матеріалу

Всі існуючі формати серіалізації даних для передачі в комп'ютерних мережах, так як і протоколи, діляться на текстові та бінарні. Найпопулярнішими текстовими форматами є JSON та XML. Прикладом текстового протоколу прикладного рівня є HTTP. Особливістю текстових форматів та протоколів даних є те, що вони більш зрозумілі для інженерів, програмістів, які працюють з ними.

У теорії для відлагодження та виявлення помилок в них не потрібні спеціальні інструменти для декодування даних. Однак платою за це є більші вимоги до обчислювальних ресурсів. Бінарні формати (наприклад BSON) та протоколи (HTTP2), на противагу текстовим, є менш вимогливими до ресурсів.

Також варто зазначити, що текстові формати можуть використовуватись в тому числі і для передачі бінарних даних. Зокрема, і в JSON, і в XML є обхідні шляхи, як це зробити, хоча й з додатковими затратами обчислювальних ресурсів.

Перш ніж переходити до бінарних форматів, розглянемо спочатку текстові формати серіалізації – XML та JSON.

XML – Extensible Markup Language. Формат описує клас об'єктів даних, що називаються XML-документами, і частково описує поведінку комп'ютерних програм, які їх обробляють. XML – це обмежена форма SGML, стандартної мови розмітки ISO-8879. Документи XML є сумісними з документами SGML.

Документи XML складаються з блоків, які називаються сутностями, і можуть містити як структуровані, так і неструктуровані дані. В документі присутні як розмітка даних (теги та атрибути), так і самі символьні дані. Тег кодує схему зберігання документа та логічну структуру.

XML проектувався з урахуванням наступних вимог:

– XML повинен бути сумісним із SGML.

– Простота створення програм, які обробляють XML-документи.

– Кількість необов'язкових функцій у XML повинна бути збережена до абсолютного мінімуму, в ідеалі – до нуля.

– Документи XML повинні бути чіткими та зрозумілими.

– Формат XML повинен бути підготовлений швидко.

– Формат XML повинен бути формальним та лаконічним.

– Порядок та точність розміщення даних в XML має мінімальне значення.

Кожен XML-документ містить один або кілька елементів, межі яких обмежуються початковими тегами та кінцевими тегами, а для порожніх елементів – тегом порожнього елемента. Кожен елемент має тип, ідентифікований за назвою, який іноді називають його «загальним ідентифікатором» (GI), і може мати набір атрибутів. Кожна специфікація атрибута має назву та значення [25].

JSON – JavaScript Object Notation. Це текстовий незалежний формат, який використовує конвенції мов програмування сімейства С.

JSON побудований на двох структурах:

– Колекції пар «назва/значення». У різних мовах програмування колекція реалізується як об'єкт, запис, структура, словник, хеш-таблиця, або асоціативний масив.

– Упорядкований список значень. Реалізується як масив, вектор, список або послідовність.

Це – універсальні структури даних, які підтримуються завжди у тій чи іншій формі.

JSON описує наступні структурні елементи:

1) Об'єкт – це неупорядкований набір пар назва/значення. Об'єкт починається із символу «{« і закінчується символом «}». За кожною назвою слідує символ «:». Кожна пара назва/значення розділяється комою.

2) Масив – це упорядкована сукупність значень. Масив починається із символу «[« і закінчується символом «]». Значення розділяються комою.

Значенням може бути текст у подвійних лапках, число, об'єкт, масив або спеціальні зарезервовані слова: true, false і null. Ці структури можуть бути вкладені одна в одну.

Текстовий рядок являє собою послідовність символів Unicode, обов'язково обернуті подвійними лапками [11].

Оскільки XML та JSON – текстові формати, то їх легко порівняти (табл. 1).

Таблиця 1 – Порівняння XML та JSON

	XML	JSON
Приклад (compressed)	`<falls><fall fall=”fell” type=«point»><coordinates><latitude>71.8000</latitude><longitude>32.1000</longitude></coordinates><info><mass>30000</mass><name>Agen</name><recclass/></info></fall></falls>`	`[{“fall”:“fell”,“type”: “point”,“coordinates”:{“latitude”:71.8000,“longitude” : 32.1000},“info”:{“mass”:30000,“name”:“Agen”,“recclass”: null}}]`

Приклад (Human-readable)	`<falls>`
	`  <fall fall=”fell” type=”point”>`
	`  <coordinates>`
	`  <latitude>71.8000</latitude>`
	`  <longitude>32.1000</longitude>`
	`  </coordinates>`
	`  <info>`
	`  <mass>30000</mass>`
	`  <name>Agen</name>`
	`  <recclass/>`
	`  </info>`
	`  </fall>`
	`</falls>`
		`[{`
		`  “fall” : “fell”,`
		`  “type” : “point”,`
		`  “coordinates”: {`
		`  “latitude”: 71.8000,`
		`  “longitude”: 32.1000`
		`  },`
		`  “info” : {`
		`  “mass”: 30000,`
		`  “name”: “Agen”,`
		`  “recclass”: null`
		`  }`
		`}]`

Продовження таблиці 1

	XML	JSON
Розмір (байтів)	208	143
Розмір (GZIP 1.6 amd64, байтів)	148	132

Оскільки текстові формати кодування не є надто ефективними, альтернативами до них являються бінарні формати, що також можуть підтримувати довільні структури та передачу схеми даних поряд із самими даними [26].

Далі розглянемо наступні формати: BSON, Smile, MessagePack.

5. Бінарні формати із серіалізацією схеми даних

BSON – Binary JSON. Це специфікація двоетапної серіалізації документів, подібних до JSON. Як і JSON, BSON підтримує багаторівневі структури документів та масивів. BSON також містить розширення, які дозволяють передавати типи даних, які не входять до специфікації JSON. Наприклад, BSON має тип для дат та тип для бінарних даних.

BSON можна порівнювати з бінарними форматами обміну, наприклад Protocol Buffers, але, на відміну від останнього, він більш гнучкий. Однак BSON має накладні витрати, оскільки схема даних передається разом із самими даними.

Теоретично BSON може бути альтернативою формату JSON та використовуватись у протоколі HTTP/2, який є бінарним.

BSON розроблено з урахуванням наступних вимог:

1) Легковісність. Зменшення накладних витрат до мінімуму є важливим для будь-якого формату даних, особливо, якщо він використовується для передачі у комп'ютерних мережах.

2) Зручність. BSON є простим форматом з простою структурою, що дозволяє легко маніпулювати структурними одиницями.

3) Ефективність. Кодування даних в BSON та декодування з BSON можуть виконуватися дуже швидко в більшості мовах.

Базові типи BSON наведені нижче в таблиці 2.

Таблиця 2 – Базові типи

byte	1 байт (8 біт)
int32	4 байти (ціле число зі знаком)
int64	8 байтів (ціле число зі знаком)
uint64	8 байтів (ціле число без знаку)
double	8 байтів (64-bit IEEE 754-2008)
decimal128	16 байтів (128-bit IEEE 754-2008)

Вищенаведений приклад пакета у форматі BSON займає 157 байтів без GZIP та 151 з GZIP.

Закодована структура в BSON:

```
0x00:  91  00  00  00  03  30  00  89  00  00  00  02  66  61  6c  6c
.....0......fall
0x10:  00  05  00  00  00  66  65  6c  6c  00  02  74  79  70  65  00
.....fell..type.
0x20:  06  00  00  00  70  6f  69  6e  74  00  03  63  6f  6f  72  64
....point..coord
0x30:  69  6e  61  74  65  73  00  2a  00  00  00  01  6c  61  74  69
inates.*....lati
0x40:  74  75  64  65  00  33  33  33  33  33  f3  51  40  01  6c  6f
tude.33333.Q@.lo
0x50:  6e  67  69  74  75  64  65  00  cd  cc  cc  cc  cc  0c  40  40
ngitude.......@@
......
```

Приклад розбору перших 80 байтів (рис. 1):

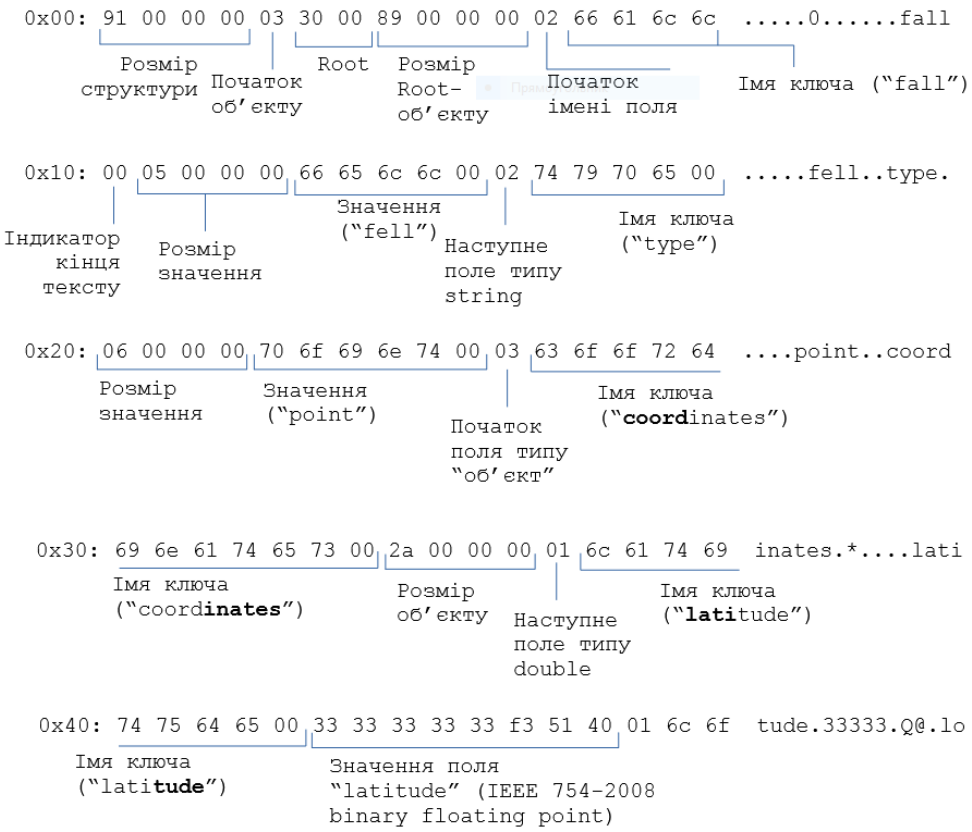

Рисунок 1 – Приклад серіалізованої структури в BSON

Можна помітити, що багато місця займають значення розмірів ключів та структур типу масиву, об'єкта, текстового рядка (int32, 4 байти на кожне поле). Враховуючи той факт, що зазвичай назви полів складаються з одного, максимум двох слів, то в результаті отримуємо overhead, який теоретично

може досягати до 100% відповідного поля в текстовому форматі JSON (якщо поле з однобайтовою назвою, то розмір поля в json: 1 символ + 2 байти на лапки + 1 байт на двокрапку). Отже, можна зробити висновок, що BSON не набагато ефективніший за текстовий аналог JSON [13].

Smile – це бінарний формат даних, еквівалент стандартного формату даних JSON.

Smile спроектований з урахуванням наступних цілей:

1) Підтримка всіх можливостей JSON.
2) Компактність для більш ефективної обробки потоку даних.
3) Ефективність у плані швидкості як при десеріалізації, так і при серіалізації.
4) Уникнення складних структур та алгоритмів.
5) Підтримка технології веб-сокетів настільки, наскільки це можливо.

Поточна версія Smile: 1.0.4. Офіційно використовується Content-Type: «application/x-jackson-smile». Потік даних починається із заголовку 0x3A 0x29 0x0A. Четвертий байт кодує наступну інформацію: номер версії, чи присутні байтові дані та тип кодування текстових даних.

Дані кодуються по секціях. Кожна секція складається з набору токенів, які формують відповідні ключі та значення. Токени використовуються у двох основних режимах: режим значення (value tokens), а також режим ключа (key tokens). Key tokens використовуються для позначення імен полів об'єктів. Довжина токена варіюється від 1 до 9 байтів. Перший байт визначає тип, а додаткові байти використовуються для специфікації в межах одного типу.

Також Smile підтримує цікаву можливість – Shared Strings. Для текстових даних, розміром менше 64 байтів, Smile застосовує кешування. Тобто, якщо в структурі, що кодується, є текстові значення, що повторюються, то фактично вони будуть закодовані лише один раз (рис. 2, табл. 3):

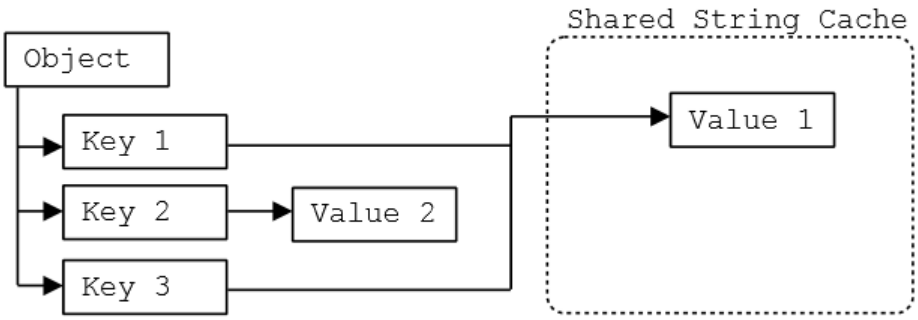

Рисунок 2 – Кеш текстових полів

Таблиця 3 – Типи даних в Smile

Empty String	Спеціальне значення для пустих рядків («»)
Simple Literals	NULL, false, true, цілі числа, числа з плаваючою точкою (32-бітні та 64-бітні)
Small ASCII, Tiny ASCII	Текстові дані ASCII

Продовження таблиці 3

Empty String	Спеціальне значення для пустих рядків («»)
Small Unicode, Tiny Unicode	Текстові дані Unicode
Small Integers	Числові дані, закодовані алгоритмом ZigZag
Misc (binary, text, structured markes)	Об’єкти, масиви, бінарні та тесктові масиви великого розміру

З табл. 3 можна бачити, що числа кодуються за допомогою алгоритму ZigZag. Це простий алгоритм, що дозволяє привести знакові типи чисел до беззнакових для уніфікації їх зберігання та передачі.

Кодування 32-бітних знакових чисел у ZigZag (табл. 4) відбувається наступним чином:

$$N_Z = (N_O << 1) \text{ XOR } (N_O >> 31) \quad (1)$$

Для 64-бітних знакових чисел:

$$N_Z = (N_O << 1) \text{ XOR } (N_O >> 63) \quad (2)$$

N_O – оригінальне значення;
N_Z – закодоване значення.

Таблиця 4 – Приклади закодованих чисел в ZigZag

Signed Int	ZigZag encoded Int
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

Нижче наведений приклад пакета закодованих даних у форматі Smile.

```
0x00: 3a 29 0a 01 fa 83 66 61 6c 6c 43 66 65 6c 6c 83
:)....fallCfell.
0x10: 74 79 70 65 44 70 6f 69 6e 74 8a 63 6f 6f 72 64
typeDpoint.coord
0x20: 69 6e 61 74 65 73 fa 87 6c 61 74 69 74 75 64 65
inates..latitude
0x30: 29 00 40 28 7c 66 33 19 4c 66 33 88 6c 6f 6e 67
).@(|f3.Lf3.long
0x40: 69 74 69 64 65 29 00 40 20 03 19 4c 66 33 19 4d  itide).@
..Lf3.M
0x50: fb 83 69 6e 66 6f fa 83 6d 61 73 73 24 07 29 a0
..info..mass$.).
0x60: ...
```

Розмір закодованих даних – 128 байтів без GZIP та 137 байтів з GZIP.

Нижче детально розглянуті перші 64 байтів серіалізованої структури (рис. 3):

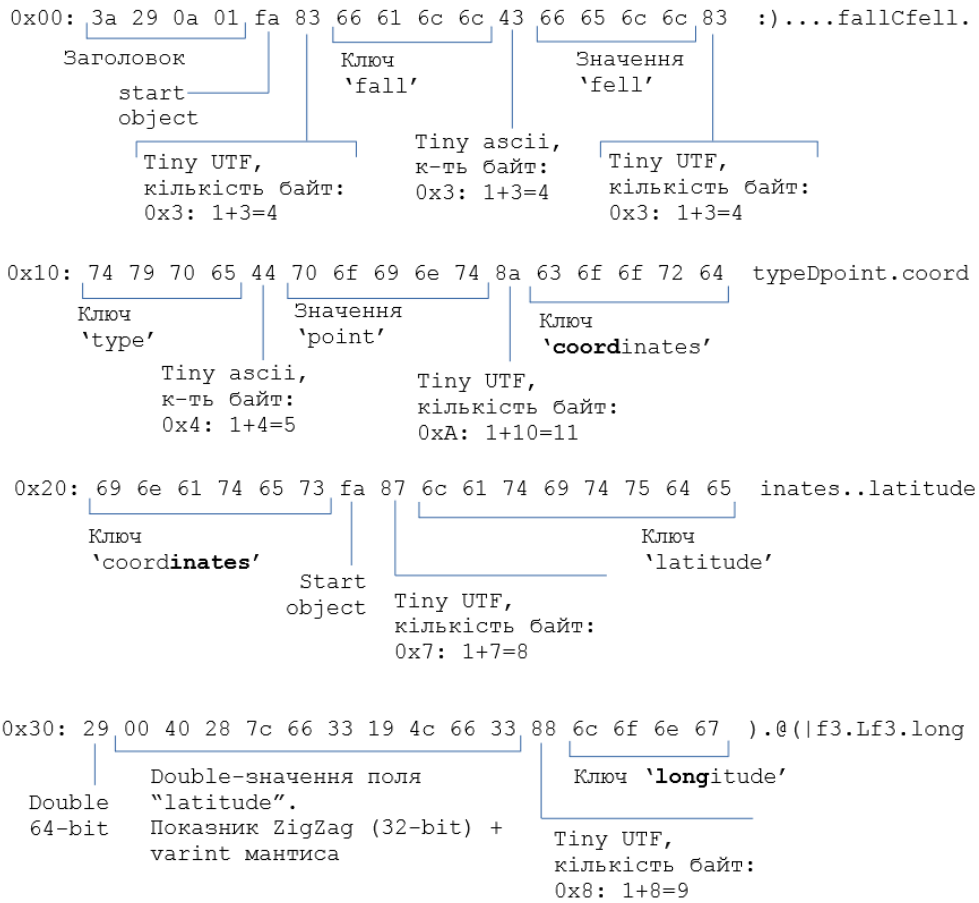

Рисунок 3 – Приклад серіалізації структури в Smile

Як можна побачити з даного прикладу, кодування більш ефективне у тому плані, що метаінформація про ключі та значення займає значно менше місця у серіалізованій структурі за рахунок використання підтипів та кодування Varint + ZigZag для чисел [14].

MessagePack – механізм серіалізації/десеріалізації об'єктів та формат обміну даними, подібний до BSON та Smile. В MessagePack є своя система типів:

1) Цілі числа – Integer.

2) NIL – значення відсутнє, відповідає NULL в мовах програмування сімейства C.

3) Boolean = true/false

4) Float – IEEE 754 (число з плаваючою точкою + спеціальні значення NaN та Infinity)

5) Raw Binary та Raw String – бінарні та текстові дані довільної довжини відповідно.

6) Групи – масиви, об'єкти та словники (пари ключ/значення).

Обмеження:

1) Числові значення обмежуються діапазоном: $\left[-2^{63};2^{64}-1\right]$.

2) Максимальна довжина об'єкта Binary: $2^{32}-1$.

3) Максимальний розмір байта об'єкта String: $2^{32}-1$.

4) Рядкові об'єкти можуть містити недійсну послідовність байтів, а поведінка десеріалізатора залежить від фактичної реалізації. Десеріалізатори повинні надавати функціональні можливості для отримання вихідного масиву байтів, щоб додатки могли вирішити, як обробляти об'єкт.

5) Максимальна кількість елементів об'єкта Array: $2^{32}-1$. Максимальна кількість асоціацій ключових значень об'єкта Map: $2^{32}-1$.

Також в MessagePack присутні оптимізації для складних об'єктів та бінарних/текстових даних довільної довжини. Фактично, кожен тип має декілька підтипів, в залежності від кількості даних. Наприклад, нижче представлені схеми кодування текстових рядків (рис. 4):

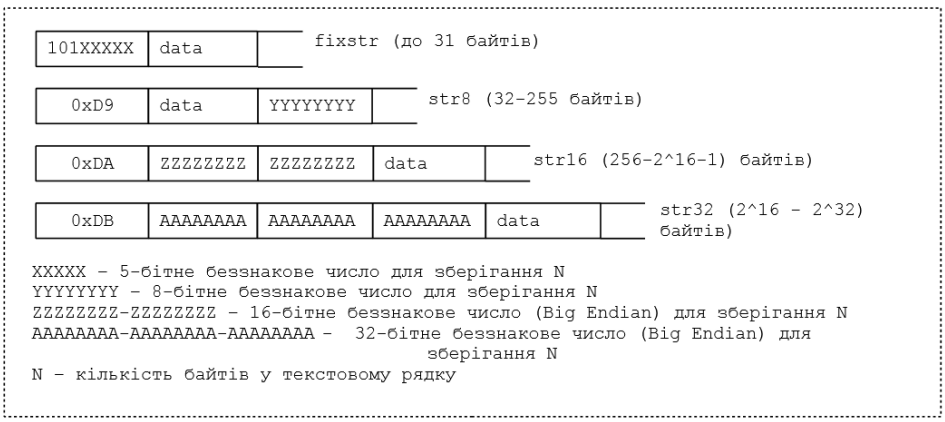

Рисунок 4 – Кодування текстових рядків в залежності від їх довжини

Аналогічні оптимізації реалізовані для чисел, масивів та словників.

Приклад, наведений вище, серіалізований за допомогою MessagePack, займає 114 байтів без GZIP та 122 байтів з GZIP.

Вигляд пакета закодованих даних:

```
0x00:  84  a4  66  61  6c  6c  a4  66  65  6c  6c  a4  74  79  70  65
..fall.fell.type
0x10:  a5  70  6f  69  6e  74  ab  63  6f  6f  72  64  69  6e  61  74
.point.coordinat
0x20:  65  73  82  a8  6c  61  74  69  74  75  64  65  cb  40  51  f3
es..latitude.@Q.
0x30:  33  33  33  33  33  a9  6c  6f  6e  67  69  74  69  64  65  cb
33333.longitide.
0x40:  40  40  0c  cc  cc  cc  cc  cd  a4  69  6e  66  6f  83  a4  6d
@@.......info..m
0x50:  ...
```

Розглянемо перші 48 байтів серіалізованої структури більш детально (рис. 5):

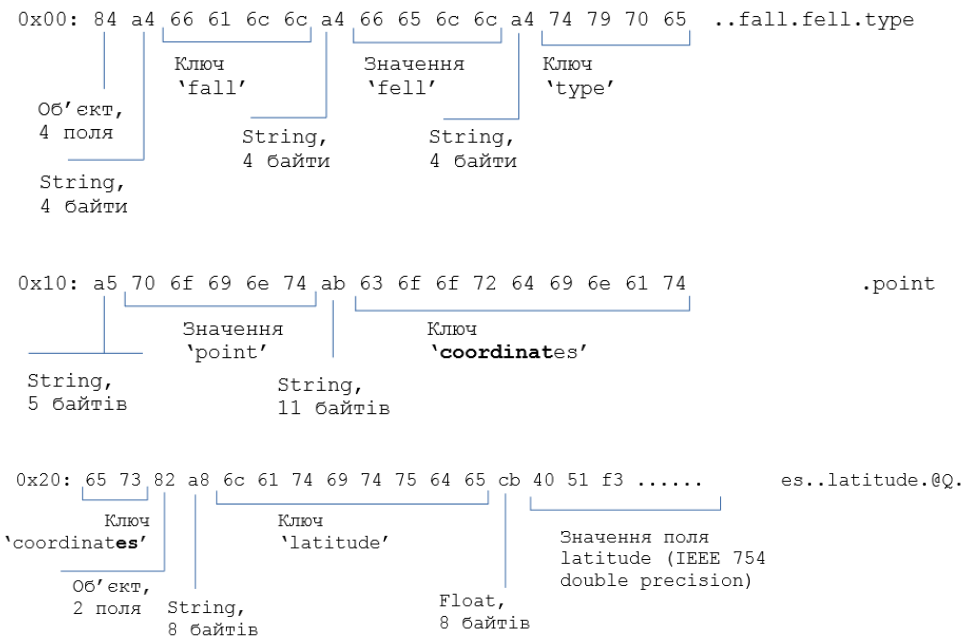

Рисунок 5 – Приклад серіалізації структури в Smile

Як і в Smile, MessagePack також використовує підтипи для більш ефективного кодування. Однак тут також помітна і відмінність від Smile: кодування об'єктів та масивів також використовує систему підтипів в залежності від розміру об'єкта/масиву. Зокрема, метаінформація для більшості простих структур (об'єкти та масиви до 8 елементів наприклад) кодуються всього-на-всього одним байтом. Числа з плаваючою точкою, на відміну від Smile, кодуються за стандартною схемою, згідно з IEEE 754 [15]. В таблиці нижче наведено порівняння BSON, Smile та MessagePack для тестового серіалізованого об'єкта.

Таблиця 5 – Розмір серіалізованих повідомлень

	BSON	Smile	MessagePack
Розмір закодованого повідомлення	157 байтів	128 байтів	114 байтів
Розмір закодованого повідомлення + GZIP	151 байтів	137 байтів	122 байтів

Далі розглянемо формати серіалізації даних, які розділяють схему даних від самих даних. Це – найефективніші формати у плані швидкості серіалізації/десеріалізації та результуючого розміру пакетів.

6. Бінарні формати без серіалізації схеми даних

Protocol Buffers – нейтральний та платформо-незалежний спосіб серіалізації та передачі структурованих даних, розроблений Google, подібний до XML, але менший, швидший та простіший. Кожне повідомлення в Protocol Buffers є невеликим логічним записом інформації, що містить серію пар ім'я – значення. Схема даних задається у файлах спеціального формату *.proto. Для кожного типу повідомлення має бути створений відповідний proto-файл. Формат файлу простий – кожен тип повідомлення має один або більше однозначно пронумерованих полів. Кожне поле має ім'я і тип значення, де типами значень можуть бути числа (цілі або з плаваючою точкою), булеві значення, текстові рядки, байти або навіть інші типи повідомлень Protocol Buffers, які дозволяють структурувати дані ієрархічно. Також можна вказати необов'язкові поля. Після створення proto-файлу, необхідно запустити компілятор Protocol Buffers для створення класів доступу до даних. Вони забезпечують прості методи доступу для кожного поля (наприклад, name() і set_name ()), а також методи серіалізації/десеріалізації всієї структури.

Protocol Buffers має багато переваг над XML, JSON та іншими форматами для серіалізації структурованих даних. Зокрема, за заявою розробників Protocol Buffers:

- займає до 3–10 разів менше місця;
- працює до 20–100 разів швидше;
- менш двозначний;
- дозволяє створювати класи доступу до даних, простих у використанні програмними засобами.

Тим не менш, робота з Protocol Buffers більш складна, ніж з іншими форматами. По-перше, необхідно створити Proto-файл:

```
syntax = "proto2";

package proto;
option java_package = "com.github.romychab.randomsets.proto";
option java_outer_classname = "ProtoFall";

message Coordinates {
  required double latitude = 1;
  required double longitude = 2;
}

message Info {
  required int32 mass = 1;
  required string name = 2;
  optional string recclass = 3;
}

message Fall {
  enum FallType {
    FELL = 1;
    DESTROYED = 2;
    UNKNOWN = 3;
  }
  required string fall = 1;
  required FallType type = 2;
  optional Coordinates coordinates = 3;
```

```
  required Info info = 4;
}

message Falls {
  repeated Fall fall = 1;
}
```

Далі необхідно встановити компілятор protoc та підключити платформо-залежні бібліотеки. Наприклад, для Java та системи Maven необхідно додати залежність:

```
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.6.1</version>
</dependency>
```

Потім необхідно скомпілювати Proto-файл, наприклад наступним чином:

```
$ protoc --java_out=../src/main/java falls.proto
```

В результаті отримаємо Java-клас для серіалізації/десеріалізації даних.

Вигляд пакета закодованих даних:

```
0x00:  0a  28  0a  04  66  65  6c  6c  10  01  1a  12  09  33  33  33
.(..fell.....333
0x10:  33  33  f3  51  40  11  cd  cc  cc  cc  cc  0c  40  40  22  0a
33.Q@.......@@".
0x20: 08 b0 ea 01 12 04 41 67 65 6e                    ......Agen
```

Кількість байтів: 42. Кількість байтів з GZIP: 58.

Однак, варто зазначити, що розмір зкомпільованого класу Protocol Buffers для одного типу тестового повідомлення – 156 Кб.

Protocol Buffers використовує подвійне кодування цілих чисел: ZigZag + VarInt. Про ZigZag було вже згадано вище. Загалом схема кодування виглядає наступним чином (рис. 6):

Рисунок 6 – Подвійне кодування цілих знакових чисел

Типів даних в Protocol Buffers небагато. На нижньому рівні існує всього 6 типів Varint (цілі числа), 64-bit та 32-bit (дробові числа), Length-delimited (текстові рядки, масиви байтів), Start Group та End Group (зараз помічені як deprecated, використовувались для позначення груп). При кодуванні повідомлення формується потік даних, що складається з послідовних полів:

Кожен ключ (Key) – це тип Varint, значення якого формується наступним чином:

$$\text{Key} = (F_N << 3) \mid \text{type}$$

Кодування значень (Value) залежить від типу. Наприклад, байтові масиви та текстові рядки кодуються примітивною структурою: довжина (Varint) + дані (Bytes) [16].

Flat Buffers – формат та метод серіалізації даних, подібний до Protocol Buffers, але спеціально розроблений для систем з дійсно високим навантаженням. Його особливість полягає в оптимізованому використанні системних ресурсів – як часу на серіалізацію/десеріалізацію, так і на розмір використовуваної пам'яті.

Flat Buffers надає доступ до даних без процесу серіалізації/десеріалізації. Всі ієрархічні дані відразу представляються в оригінальному бінарному буфері без проміжної трансформації в об'єкти чи платформо-залежні представлення. При цьому також підтримується як міграція структур даних, так і зворотна сумісність.

Наочно різницю між Protocol Buffers та Flat Buffers можна представити у вигляді схем (рис. 7):

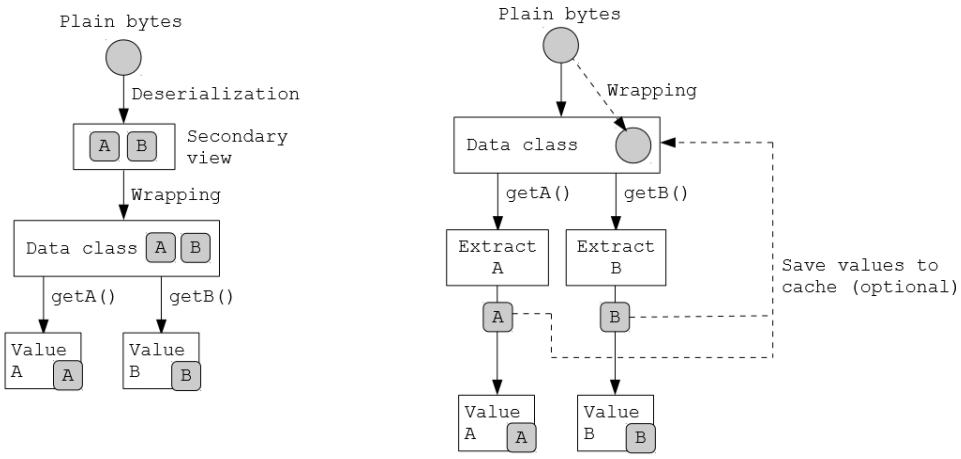

Рисунок 7 – Десеріалізація даних в Protocol Buffers (зліва) та Flat Buffers (справа)

Flat Buffers економніший у плані використання ресурсів процесора та пам'яті. Для своєї роботи Flat Buffers необхідний лише буфер байтів, тобто алокація пам'яті виконується лише один раз.

Не створюються проміжні об'єкти, а самі значення конвертуються у потрібний вигляд лише при безпосередньому запиті самих даних. Це економить процесорний час, оскільки часто користувачеві необхідна лише частина даних з отриманого пакета. Дані, які користувач не використовує, не будуть оброблюватись та декодуватись. Тому Flat Buffers також підходить для використання з технологією mmap та для потокової передачі.

Також Flat Buffers має підтримку необов'язкових полів, що дозволяє модифікувати повідомлення від версії до версії. Це означає, що при деяких змінах у протоколі передачі даних з часом простіше організувати зворотну

сумісність з попередніми версіями. Підтримується міграція як в сторону додавання додаткових структур, так і їх видалення з пакетів протоколу.

Як і у випадку з Protocol Buffers, Flat Buffers використовує кодогенерацію для створення класів доступу до серіалізованих даних.

Бінарні файли компілятора Flat Buffers офіційно поставляються лише для ОС Windows. Для інших платформ необхідно завантажити вихідні тексти, систему cmake та скомпілювати самостійно:

```
$ sudo apt-get install cmake
$ git clone https://github.com/google/flatbuffers.git
$ cd flatbuffers
$ cmake -G «Unix Makefiles» -DCMAKE_BUILD_TYPE=Release
```

Аналогічно до Protocol Buffers, необхідно описати схему повідомлень у спеціальному файлі. Зазвичай йому дають розширення .FBS:

```
namespace com.github.romychab.randomsets.flat;

enum FlatFallType:byte { FELL = 0,  DESTROYED = 1,  UNKNOWN = 2 }
struct FlatCoordinates { latitude:double;  longitude:double; }

table FlatInfo {
  mass:int;
  name:string;
  recclass:string;
}

table FlatFall {
  fall:string;
  type:FlatFallType;
  coordinates:FlatCoordinates;
  info:FlatInfo;
}

table FlatFalls { falls:[FlatFall]; }

root_type FlatFalls;
```

Генерація класів для Java:

```
$ flatc --java flat.fbs
```

Вигляд пакета закодованих даних:

```
0x00000:  08  00  00  00  00  00  00  00  01  00  00  00  10  00  00  00
................
0x00010:  0c  00  20  00  04  00  00  00  0c  00  08  00  0c  00  00  00   ..
............
0x00020:  1c  00  00  00  2c  00  00  00  33  33  33  33  33  f3  51  40
....,...33333.Q@
0x00030:  cd  cc  cc  cc  cc  0c  40  40  00  00  00  00  04  00  00  00
......@@........
0x00040:  66  65  6c  6c  00  00  00  00  08  00  0c  00  04  00  08  00
fell............
0x00050:  08  00  00  00  30  75  00  00  04  00  00  00  04  00  00  00
....0u..........
0x00060: 41 67 65 6e 00 00 00 00                          Agen....
```

Кількість байтів без GZIP – 104. Кількість байтів з GZIP – 85.

Класи серіалізації/десеріалізації, що генеруються в FaltBuffers, значно менші за розміром на порядок, аніж відповідні класи ProtocolBuffers. Тим не менш, робота з ними є більш складною, оскільки вимагає напряму маніпулювати числовими ідентифікаторами полів та структур [17].

Наочно різницю між формуванням структур у Protocol Buffers та FlatBuffers можна показати за допомогою вихідних текстів (табл. 6).

Таблиця 6 – Порівняння вихідних текстів для формування структур даних в Protocol Buffers та Flat Buffers

Protocol Buffers	Flat Buffers

`Falls toProtocolBuffers() {`
 	
` Info info = Info.newBuilder()`
`  .setMass(this.info.mass)`
`  .setName(this.info.name)`
 	
`.setRecclass(this.info.recclass)`
`  .build();`
	`Falls toFlatBuffers() {`
	` FlatBufferBuilder builder = new`
	`  FlatBufferBuilder();`
	
	` int nameOffset =`
	`builder.createString(this.info.name);`
	` Info.startInfo(builder);`
	` Info.addName(builder,`
	`nameOffset);`
	` Info.addMass(builder,`
	`this.info.mass);`
	` int flatInfoOffset =`
	`Info.endInfo(builder);`

`Coordinates coordinates =`
`  Coordinates.newBuilder()`
 	
`.setLatitude(this.coordinates.lat)`
 	
`.setLongitude(this.coordinates.lon)`
`  .build();`
` Fall fall = Fall.newBuilder()`
`  .setType(Fall.FallType.FELL)`
`  .setFall(this.fall)`
`  .setInfo(info)`
`  .setCoordinates(coordinates)`
`  .build();`
	`int fallOffset =`
	
	`builder.createString(this.fall);`
	` Fall.startFall(builder);`
	` Fall.addCoordinates(`
	`  builder,`
	`  Coordinates.createCoordinates(`
	`  builder,`
	`  this.coordinates.lat,`
	`  this.coordinates.lon`
	`  ));`
	` Fall.addInfo(builder,`
	`flatInfoOffset);`
	` Fall.addType(builder,`
	`FallType.FELL);`
	` Fall.addFall(builder,`
	`fallOffset);`
	` int flatFallOffset =`
	`Fall.endFall(builder);`

` return Falls.newBuilder()`
`  .addFall(fall)`
`  .build();`
`}`
	` int flatFallsOffset =`
	`  Falls.createFallsVector(builder,`
	`new int[]`
	`  { flatFallOffset} );`
	
	` builder.finish(flatFallsOffset);`
	` byte[] data =`
	`builder.sizedByteArray();`
	` return Falls.getRootAsFalls(`
	`  ByteBuffer.wrap(data));`
	`}`

Підведемо основні тези та підсумки на основі розглянутого матеріалу та проведених досліджень (табл. 7).

Таблиця 7 – Порівняльна характеристика розглянутих форматів серіалізації структурованих даних

Результати серіалізації тестової структури даних						
	Тип	Схема	Розмір закодованої структури без GZIP (байт)	Розмір закодованої структури з GZIP (байт)	Розмір проміжних структур (Кб)	Примітки
XML	Текстовий	Нестрога	208 (max)	**148** (max)	-	
JSON	Текстовий	Нестрога	143	**132**	-	
BSON	Бінарний	Нестрога	157	**151**	-	Структура аналогічна з JSON, але кодується в бінарний формат
Message Pack	Бінарний	Нестрога	**114**	122	-	
Smile	Бінарний	Нестрога	**128**	137	-	
Protocol Buffers	Бінарний	Строга	**42** (min)	58 (min)	156 (Компілятор protoc)	
Flat Buffers	Бінарний	Строга	104	**85**	12.9 (flatc)	Швидке кодування даних

Висновки

1) Текстові структури є найбільш неефективними у плані розміру закодованих даних.

2) Найбільш ефективним форматом кодування малих повідомлень є Protocol Buffers, однак платою за це є значне збільшення розміру скомпільованого програмного забезпечення.

3) Якщо необхідно досягти максимальної пропускної здатності каналу передачі даних, то необхідно використовувати бінарні протоколи зі строгою схемою. В цьому випадку коефіцієнт відношення об'єму корисних даних до загального об'єму повідомлення при передачі є найбільш високим.

4) Застосування існуючих бінарних протоколів зі строгою схемою є більш складним, оскільки необхідно підтримувати міграцію структури, зворотну сумісність, а також встановлювати спеціальне програмне

забезпечення для кодогенерації проміжних класів для кожної використовуваної платформи.

5) В той же час, недоліку п.5 позбавлені текстові формати XML та JSON. Однак, вони є найбільш неефективними форматами серіалізації структурованих даних – серіалізовані повідомлення великі за розміром та містять надлишкову інформацію.

6) Бінарні дані з нестрогою схемою можна вважати «компромісним» рішенням між складністю в реалізації та ефективністю.

7) У Smile та MessagePack використовуються механізм підтипів для більш ефективного кодування даних. Як було показано у статті, це ускладнює структуру даних, однак в результаті розмір серіалізованих даних менший, ніж в аналогах без механізму підтипів. Теоретично, можна інтегрувати цей механізм у бінарні протоколи зі строгою схемою, що дозволить формувати структури меншого розміру.

8) Flat Buffers серіалізує/десеріалізує тільки ті структури даних, що безпосередньо використовуються користувачем, що значно прискорює роботу програмного забезпечення. Тим не менш, Flat Buffers є механізмом кодування дуже низького рівня і з ним складно працювати. Пропонується залучити механізм буферизації з Flat Buffers, при цьому залишаючи абстракції формування структур даних Protocol Buffers. Це може знизити швидкість формування даних, але в той же час і значно підвищить швидкість їх серіалізації та передачі по мережевих каналах, що приведе до економії сумарного часу, необхідного на передачу даних між хостами.

СПИСОК ЛІТЕРАТУРИ

1. L. Roberts. Beyond Moore's law: Internet growth trends / Computer, 2000. – Vol. 33, Iss. 1, P. 117–119.
2. Sahm Kim. Forecasting internet traffic by using seasonal GARCH models / Journal of Communications and Networks, 2011. – Vol. 13, Issue 6, P. 621–624.
3. Cisco Visual Networking Index: Forecast and Methodology: 2016–2021 / Cisco VNI, 2017.
4. H. Hamidi. The role of the Internet of Things in the improvement and expansion of business / M. Jahanshahifard / Journal of Organizational and End User Computing, 2018. – Vol. 30, Issue 3, P. 22–24.
5. Apache CouchDB Documentation. JSON Structure Reference [Electronic resource]. – Mobile access: http://docs.couchdb.org/en/stable/json-structure.html
6. MongoDB for Giant Ideas. Json & BSON [Electronic resource]. – Mobile access: www.mongodb.com/json-and-bson
7. SQLite Consortium. SQLite JSON1 Extension [Electronic resource]. – https://www.sqlite.org/json1.html
8. The Json Data Type – MySQL Reference Manual [Electronic resource]. – Mobile access: https://dev.mysql.com/doc/refman/8.0/en/json.html
9. User Datafram Protocol: Internet Standard, RFC-768 / ISI, 1980
10. Transmission Control Protocol: Internet Standard, RFC-793 / Information Sciences Institute University of Southern California, 1981
11. ECMA-404 The JSON Data Interchange Standard – Ecma International, 2017 [Electronic resource]. – Mobile access: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
12. Extensible Markup Language – World Wide Web Consorcium [Electronic resource]. – Mobile access: https://www.w3.org/XML/

13. BSON Spec. BSON (Binary JSON): Specification [Electronic resource]. – Mobile access: http://bsonspec.org/spec.html
14. FasterXML. Smile Data Format [Electronic resource]. – Mobile access: https://github.com/FasterXML/smile-format-specification
15. Sadayuki Furuhashi. MessagePack [Electronic resource]. – Mobile access: https://msgpack.org/
16. Google Developers. Protocol Buffers: Developer Guide [Electronic resource]. – Mobile access: https://developers.google.com/protocol-buffers/docs/overview
17. GitHub IO. Flat Buffers Internals [Electronic resource]. – Mobile access: https://google.github.io/flatbuffers/flatbuffers_internals.html
18. Srđan Popić. Performance evaluation of using Protocol Buffers in the Internet of Things communication / Dražen Pezer, Bojan Mrazovac, Nikola Teslić / 2016 International Conference on Smart Systems and Technologies (SST), 2016, P. 261–265.
19. Heejung Kim. The upcoming new standard HTTP/2 and its impact on multi-domain websites / Jongseok Lee, Ikhyun Park, Hyungkyung Kim, Dong-Hoon Yi, Taesung Hur / 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2015, P. 530–533.
20. Kazuaki Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats / Digital Information and Communication Technology and it's Applications (DICTAP), 2012, P. 177–182.
21. K. Chiu. A binary XML for scientific applications / T. Devadithya, Wei Lu, A. Slominski / First International Conference on e-Science and Grid Computing, 2005, P. 343–350.
22. M. Isenburg. Coding with ASCII: compact, yet text-based 3D content / J. Snoeyink / Proceedings. First International Symposium on 3D Data Processing Visualization and Transmission, 2002, P. 609–616.
23. Maxim Ya. Afanasev. Performance evaluation of the message queue protocols to transfer binary JSON in a distributed CNC system / Yuri V. Fedosov, Anastasiya A. Krylova, Sergey A. Shorokhov / IEEE 15th International Conference on Industrial Informatics (INDIN), 2017, P. 357–362.
24. Jianhua Feng. Google protocol buffers research and application in online game / Jinhong Li. / IEEE Conference Anthology (original: 13Th IEEE Joint International Computer Science and Information Technology Conference), republished in 2013, P. 1–4.
25. World Wide Web Consorcium. Extensible Markup Language [Electronic resource] – Mobile access: https://www.w3.org/XML
26. Kazuaki Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats / 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP), 2012, P. 177.

Стаття надійшла до редакції 07.09.2018.