УДК 004.65

Д.О. Вітковський, С.Ф. Теленик

БАЗА ДАНИХ ПУБЛІЧНИХ РЕАКТИВНИХ ФУНКЦІОНАЛЬНИХ СИГНАЛІВ

Анотація: У статті запропоновано і досліджено варіант побудови структури бази даних та системи керування базами даних для реалізації можливостей функціональних реактивних сигналів на рівні глобальної бази даних, що дозволить збільшити можливості міжсервісної комунікації. Метою дослідження є розроблення моделі системи керування базами даних та клієнтських бібліотек, а також вивчення доцільності побудови даної системи. Дослідження виконане шляхом розроблення часткового прототипу системи керування базами даних та бібліотеки для фреймворку SolidJS. У результаті отримано набір алгоритмів для моделі базової реалізації системи керування базами даних та бібліотеки-клієнта. Подальші дослідження будуть спрямовані на розроблення додаткового функціоналу, насамперед технологій статичної типізації та збереження похідних сигналів у базі даних.

Ключові слова: база даних, реактивне програмування, сигнали, функціональне програмування, ефекти, події, черги повідомлень.

ВСТУП

Уже достатньо тривалий час існує підхід до побудови архітектури системи на основі подій – коли щось стається в одному місці системи, про це надсилається повідомлення, яке можна обробити у іншому місці системи незалежно. Архітектура, керована подіями, часто використовується при розробленні систем на основі мікросервісів, а також систем інтернету речей.

Разом з цим, у останні роки набули популярності так звані «сигнали» – поведінковий шаблон з функціонального реактивного програмування, що застосовується, наприклад, під час розроблення користувацьких веб-інтерфейсів. Суть шаблону полягає у наступному: значення, що відповідає за стан сторінки інтерфейсу огортається у «сигнал», і коли воно міняється, то оновлюються лише ті частини сторінки, які залежать від нього. Це дозволяє покращити швидкодію інтерфейсу шляхом зменшення кількості роботи що має бути виконана для його оновлення.

Оскільки сьогодні часто виникає потреба у побудові систем з великою кількістю взаємодіючих компонентів, пов’язаних між собою у мережі, або систем із великою кількістю синхронізованих клієнтів, то набуває актуальності проблема ефективної комунікації між складовими таких систем та коректної синхронізації їх станів.

Одним зі способів, що може допомогти це вирішити, може бути об’єднання підходів архітектури, керованої подіями, та функціональних реактивних сигналів у

© 0 0' ." 0 0' "

 ISSN 1560-8956

вигляді бази даних, що виступає джерелом правди. Завдяки цьому синхронізацію стану усіх клієнтів можливо звести до точкової зміни стану одним з клієнтів, про що решта клієнтів буде автоматично повідомлена, завдяки відповідному протоколу гарантованої доставки.

1. Проблематика роботи

Розподілена система може складатися із великої кількості пов’язаних компонентів, комунікація між якими потребує коректної синхронізації спільного стану системи.

Постає питання, який спосіб буде найкращим для забезпечення комунікації між мікросервісами? Найбільш простим способом є REST API [1]. Наскільки ефективно він може бути застосований, залежить від особливостей створюваної системи, але досвід показує, що найчастіше це порушує автономність та ізольованість сервісів: вони мають знати про функціональність та API один одного.

Іншим мінусом використання такого підходу є те, що сервіс, із яким відбувається комунікація за допомогою REST, може бути недоступним й ми не можемо точно знати коли відновиться доступ. Можна скасувати транзакцію, що буде песимістичним результатом (і потребує окремої організації API), або зберегти запит у певній черзі, щоб виконати його пізніше. При цьому ця черга має бути стійкою (англ. persistent) і, бажано, розподіленою, аби її вміст не був втрачений у разі збою.

Способи, які використовуються для виправлення цих недоліків наразі – це використання архітектури, керованої подіями, а також систем повідомлень, у якості окремого сервісу, що поєднує компоненти розподіленої системи.

У той час, як керування подіями системи може відбуватися у межах одного компоненту системи та не вимагати великих зусиль розробника, системи повідомлень призначені для керування кількома непов’язаними компонентами, що ускладнює загальну методику розробки системи. Системи повідомлень мають ряд особливостей [1].

По-перше, після відправки повідомлення, ми не знаємо, чи було воно оброблено, навіть, якщо відправка пройшла успішно.

По-друге, для комунікації та ефективного зберігання повідомлень у файловій системі, має використовуватися бінарне кодування даних, а не звичне текстове кодування.

По-третє, оскільки сервіси мають бути незалежними, а система повідомлень не має знати про їхню реалізацію, для опису схеми повідомлень використовують окремі мови визначення інтерфейсу (англ. interface definition language, IDL). Якщо є потреба змінити схему повідомлень, то система перестає розуміти повідомлення, що зберігалися у ній до змін.

У той же час, системи повідомлень не призначені для зберігання даних, що не є повідомленнями, такі як дані, що зазвичай зберігаються у базах даних. Різні СКБД,

такі як Redis [2], Valkey [4], Supabase [5], реалізують механізм повідомлення про зміни у даних [3, 6], та використовують для цього окремі канали, на які є можливість підписатися для відслідковування змін. При цьому, в основному, СКБД не переймаються гарантією доставки, а у разі збою, усі повідомлення про зміни, що не були отримані та опрацьовані, буде втрачено, оскільки, в загальному випадку, не зберігаються жодною зі сторін.

В такому разі, якщо виникає необхідність зберігати зміни, доступними варіантами вирішення питання є лише збереження даних у вигляді часових рядів, що не завжди доречно, або реалізація сервісу-посередника, що переадресуватиме повідомлення про зміну в систему повідомлень, й гарантія доставки все ще відсутня, бо повідомлення, які не отримав посередник, теж буде втрачено. Винятком є СКБД MongoDB [7, 8], що деякий час зберігає журнал операцій потоку змін (англ. change stream oplog) на диску.

Для багатьох задач така складність не є необхідною, але надійність усе ще може бути важливою. Автори статті поставили за мету запропонувати варіант дати сучасну і ефективну відповідь на це питання, поєднавши спеціально спроектовану базу даних та клієнтську бібліотеку-компаньйона, таким чином інтегруючи в рамках системи концепцію архітектури, керованої подіями, та функціональні реактивні сигнали.

2. Аналіз існуючих підходів до вирішення задачі

Найпростішим способом комунікації між сервісами залишається REST API.

У REST API є й інші недоліки, окрім згаданих раніше, як підходу для повідомлення сервісів про транзакцію [1]:

- Сервіс-відправник повинен знати усіх одержувачів повідомлення, що змушує повертатися до його реалізації кожний раз коли додається новий сервіс, або змінюється контракт існуючого;
- Складно реалізувати гарантовану доставку в разі недоступності сервісу;
- Оскільки загальна операція має бути атомарною, потрібно мати API як для самої відправки повідомлення, так і для його відкату у разі помилки, що вимагає додаткових обмежень та гарантій й загалом ускладнює кожен мікросервіс.

Для вирішення цих питань потрібен підхід, який дозволить зробити мікросервіси ізольованими, прибравши тісний зв’язок між ними, та надати засіб гарантованої доставки, отримання й зберігання повідомлень про транзакцію/зміну стану.

Ці вимоги задовольняє такий вид сервісів як «система повідомлень» (англ. messaging system). Існують різні реалізації такого виду поведінки: StormMQ, RabbitMQ [9], Apache Kafka й інші.

Ці системи мають одне призначення – пересилати повідомлення між відправником і одержувачем. При цьому, ці системи повідомлень надають різні

 ISSN 1560-8956

варіанти доставки: якщо одержувач один (англ. point-to-point), якщо одержувачів багато (англ. publisher-subscriber), якщо не важливо хто саме обробить повідомлення, але таких споживачів декілька (англ. competing consumers) тощо.

Загалом, системи повідомлень займаються лише доставкою повідомлень у надійний спосіб. У їх обов’язки не входить зберігання даних, що не є повідомленнями. Так RabbitMQ дозволяє реалізувати різні стратегії адресації повідомлень [10]:

- Point-to-Point – один споживач;
- Worker Queues (шаблон competing consumers) – розподіл задач між робочими процесами – хто перший прочитав, лише той повідомлення й отримає;
- Publish/Subscribe – повідомлення буде отримано одночасно кількома споживачами;
- Routing – отримання повідомлень відбувається вибірково
- Topics – отримання повідомлень відбувається відповідно до шаблону (теми, англ. topic);
- RPC (remote procedure call) – шаблон запит/відповідь: споживач, після отримання повідомлення, відповідає відправнику результатом.

Ці стратегії дозволяють реалізувати різноманітні сценарії використання.

Серед СКБД, які надають певну гарантію доставки є MongoDB [7]. Спосіб, яким відбувається повідомлення про зміни у даних, називається «потоки змін» (англ. change streams) [8]. При цьому, повідомлення, які не було отримано, зберігаються у так званому «журналі операцій» (англ. oplog). Недоліком цього підходу є те, що, на відміну від системи повідомлень, журнал працює у режимі оброблення запитів від клієнта (англ. pull model), і якщо клієнт не встиг обробити повідомлення про зміну, вони можуть бути видалені з журналу, через його обмеженість у розмірі, що залежить від налаштувань журналу.

Щоб запобігти втраті повідомлень про зміну, автори доповіді [11] Каміна Т. та Аотані Т. розглядають підхід до стійких (англ. persistent) значень, що змінюються у часі, що полягає у використанні розробленого ними діалекту мови Java SignalJ [12] разом із базою даних часових рядів – у даному випадку це TimescaleDB, заснована на PostgreSQL.

Даний діалект надає синтаксичні конструкції та API, що дозволяють зручно працювати із функціональними реактивними сигналами та реагувати на зміну їхніх значень. Для оголошення локальних сигналів використовується ключове слово ‘signal’, при цьому до стійких сигналів додається атрибут ‘persistent’.

Таким чином, під час компіляції, SignalJ перетворює оголошення та використання стійких сигналів на SQL запити до бази даних та виклики, що дозволяють реактивно відповідати на оновлення, що надходять.

У базі даних значення представляються у вигляді таблиць із принаймні трьома полями – серійний ідентифікатор (логічний час), фактичний час створення, та саме

значення, що змінилося. Серійний ідентифікатор потрібен для синхронізації отримання змін від декількох значень.

Розглянутий API підтримує операції для агрегації даних відповідно до можливостей бази даних. Тестування ефективності роботи показало, що даний підхід може бути застосовний для багатьох випадків.

Але особливості розподілених систем вимагають розширення можливостей підходу. У праці [13] Каміна. Т та Уено С. представляють розширення реалізації SignalJ та підходу до стійких значень, що змінюються з часом, на використання у розподілених системах, та демонструють ефективність такої архітектури, шляхом реалізації простого IoT (інтернет речей, англ. internet of things) застосунку.

Щоб досягти бажаного, необхідно було зробити сигнали стійкими та розподіленими. Автори використали ту особливість реактивних систем, що більшість з них є за природою розподіленими, при цьому часто взаємодіючи з даними у вигляді часових рядів, тобто історіями оновлень значень, що змінюються з часом. Наприклад, дані з сенсорів інтернету речей можуть відслідковуватися, а оновлення таких значень можуть зберігатися у вигляді часових рядів.

Запропонована архітектура складається з пристрою розпізнавання ідентифікаторів (англ. ID resolver), екземплярів класів-сигналів та бази даних часових рядів, оснащеної реактивним функціоналом. Пристрій розпізнавання ID дозволяє незалежну конфігурацію зв’язку між екземпляром класу-сигналу та відповідною базою даних часових рядів, спираючись на його ID. Отже, це дозволяє різним сигналам бути підключеними до різних реплік бази даних розподіленої системи.

Порівняно із попередньою роботою, у даній роботі було розширено можливості компілятора SignalJ таким чином, що можна було використовувати у якості сигналів не лише примітиви, а й більш складні типи, утворюючи їх шляхом об’єднання кількох пов’язаних сигналів у один клас-сигнал (англ. signal class).

Загалом, програма мовою SignalJ компілюється у програму мовою Java, що використовує бібліотеку SignalJ під час виконання. Точніше кажучи, щоб адресуватися до проблеми розподіленості архітектури, SignalJ вже переріс можливості простої бібліотеки і натомість перетворився в платформу, що містить у своєму складі кілька сервісів.

Аби зробити розташування стійких сигналів прозорим, пристрій розпізнавання ID відслідковує зв’язок між ідентифікаторами екземплярів класу-сигналу та відповідними екземплярами бази даних. Кожний екземпляр класу-сигналу спілкується зі своєю базою даних. Він також може спілкуватися із іншими екземплярами класів-сигналів у випадку розповсюдження оновлення значень. Кілька екземплярів класів-сигналів можуть бути гетерогамними – різні складові стійкі сигнали можуть бути пов’язані із різними екземплярами бази даних. У такому випадку, кожна з баз даних не

ISSN 1560-8956

спілкується між собою напряму. Замість цього розповсюдження оновлення відбувається лише через екземпляри класів-сигналів; коли в екземпляр класу-сигналу надходить нове значення, цей екземпляр відповідальний за оновлення своєї бази даних.

Таким чином, ці підходи до побудови стійких реактивних значень (сигналів) засновані на використанні бази даних часових рядів, що є корисним, якщо важлива історія змін значень, але вимагає додаткового місця для зберігання даних, обсяги яких постійно зростають. Для реалізації розподіленого підходу потрібне існування додаткової програмної периферії для відслідковування зв’язків між екземплярами сигналів та репліками бази даних, що ускладнює загальний алгоритм роботи системи. Тим не менш, подібних складнощів важко уникнути у розподілених системах в загальному випадку, отже вони є очікуваними.

3. Підхід до побудови бази даних публічних сигналів

3.1. Термінологія

У реактивному програмуванні, «подією» називається деяке повідомлення із можливим додатковим корисним навантаженням. В такій архітектурі існують так звані «обробники подій» – процедури, що реєструються для обробки певного виду подій. Їх може бути декілька на один вид/джерело подій. В залежності від реалізації циклу подій реєстрація та виклик обробників може відбуватися у довільному або визначеному порядку, послідовно або паралельно.

У реактивному функціональному програмуванні «сигналом» називається комірка зі значенням певного типу, яке відслідковується реактивною системою. Відслідковування відбувається шляхом виконання певного «ефекту» – набору дій, що залежить від значення даного сигналу, та підписання на сигнали, що були використані під час виконання ефекту. Сигнали можна компонувати між собою, отримуючи так звані «похідні сигнали» (англ. derived signal). Похідним сигналом можна назвати навіть звичайну функцію, яка проводить обчислення на основі значень інших сигналів. Таким чином, у залежності від виду, похідні сигнали можуть не зберігати ніякі значення, а лише інструкції для їх обчислення. Підписування ефекту на похідний сигнал відбувається через підписування на складові сигнали, що були використані під час обчислення значення. Тобто, якщо складовий сигнал не був використаний під час обчислення похідного сигналу, ефект не буде викликаний при зміні значення цього сигналу.

Поняття «події» та «сигналу» є спорідненими, з точки зору поведінки: коли відбувається подія, викликається обробник цієї події; коли змінюється значення сигналу, усі «ефекти», що підписані на цей сигнал, викликаються з новим значенням.

Таким чином, можна заключити, що зміна значення сигналу є подією, а ефекти – обробниками цієї події.

Тим не менш, на відміну від звичайних обробників подій, які реєструються імперативно, ефекти – динамічні, та реагують на зміну значення сигналу лише якщо були підписані на нього, а підписування на сигнал відбувається лише якщо значення сигналу було зчитане під час попереднього виконання ефекту.

При цьому, сигнали й події найкраще працюють у тандемі: значення сигналів можна змінювати у відповідь на певні події, наприклад, натискання кнопки інтерфейсу, що може зумовити виконання ефекту, який відповідає за логіку роботи застосунку.

Можливо сформулювати кілька додаткових видів сигналів: «тригер» (англ. trigger) – сигнал, який не зберігає значення, а натомість лише сповіщає ефекти, що на нього підписані; «пам’ятка» (англ. memo) – похідний сигнал, який пам’ятає своє попереднє значення, не виконує обчислення, якщо значення сигналів-компонентів не змінилося, не зважаючи на сповіщення про оновлення, і не сповіщає ефект про оновлення, якщо нове значення рівне попередньому.

3.2. Складові системи

Дизайн системи, що пропонується у даній роботі має на меті існування наступних компонентів:

- База даних, що зберігає дані не у таблицях чи колекціях документів, а у вигляді окремих об’єктів (які можна представити у вигляді дерев) з типізованими полями (кожне поле – вузол дерева), що при оновленні надсилають повідомлення у канал, що відповідає кожній з батьківських гілок дерева – від кореня (назва об’єкту) до листку (назва поля, що змінилося). Таким чином – кожний об’єкт є (похідним) сигналом та кожне поле об’єкту є сигналом. У подальшому можливо додати підтримку зберігання сигналів інших видів, такі як похідні сигнали-функції, пам’ятки та тригери.
- Бібліотека-компаньйон, що надає абстракції для підписування на сигнали у базі даних, створення локальних сигналів та оголошення ефектів, які реагують на зміни сигналів у базі даних. Таким чином, коли один клієнт бази даних оновлює значення сигналу в базі даних, то усі клієнти, що були на це значення підписані отримують сповіщення та автоматично виконують відповідні ефекти. Виконання локальних ефектів може не вимагати очікування результатів від решти клієнтів, що робить їх існування менш залежним.

3.3. Команди СКБД та представлення даних

Дані у базі зберігаються у вигляді об’єкту з іменованими полями різного типу (рис. 1). Мають підтримуватися усі базові типи даних: булеві, числа, рядки, списки та об’єкти. Це означає, що, при наявності підписників на поле об’єкта, якщо відбувається зміна значення дочірнього поля, повідомлення про це оновлення буде отримано й підписниками на батьківські вузли.

ISSN 1560-8956

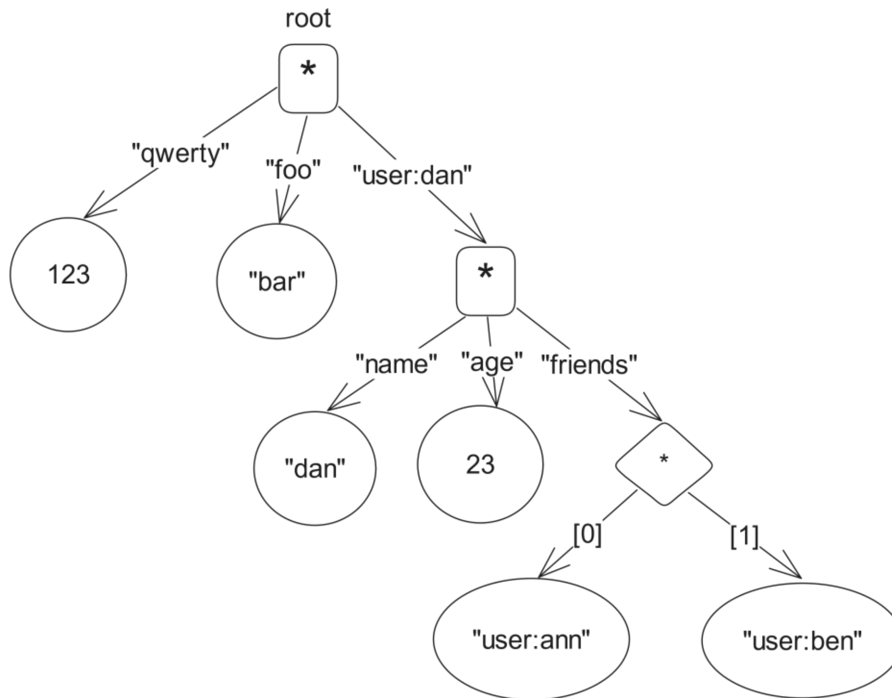

Рисунок 1. Приклад абстрактного дерева даних, що зберігається у БД

У початковій реалізації формат даних, що зберігається виглядає як JSON, а зв'язок із СКБД здійснюється за протоколом WebSocket, бо для роботи клієнта потрібен двосторонній зв'язок, а даний протокол має реалізацію на багатьох мовах програмування.

СКБД має підтримувати ряд базових команд:

- GET (UNTRACKED) – команда для отримання значення; одним з параметрів приймає байт з булевим значенням, яке вказує, чи підписувати клієнта на оновлення.
- SET – команда, яка встановлює значення за шляхом та сповіщає про це попередніх підписників значення та підписників батьківських вузлів; підписники дочірніх вузлів, що були видалені після встановлення значення сповіщаються про видалення.
- DELETE – команда, яка видаляє значення за шляхом та сповіщає про це попередніх підписників даного та дочірніх/батьківських вузлів.
- PUSH BACK – команда, яка додає значення у кінець масиву за шляхом та сповіщає про це підписників даного та батьківських вузлів.
- PUSH FRONT – команда, яка додає значення у початок масиву за шляхом та сповіщає про це підписників даного та батьківських вузлів.
- POP BACK – команда, яка прибирає значення з кінця масиву за шляхом, повертає його та сповіщає про нове значення підписників даного та батьківських вузлів, а також підписників вузлу та дочірніх вузлів що були видалені.

- POP FRONT – команда, яка прибирає значення з початку масиву за шляхом, повертає його та сповіщає про нове значення підписників даного та батьківських вузлів, а також підписників вузлу та дочірніх вузлів що були видалені.
- INSERT – команда, яка вставляє значення у масив за шляхом (не далі ніж одразу перед початком або одразу після кінця) та сповіщає про це підписників даного та батьківських вузлів.

Для реалізації ефективних операцій push/pop можна використовувати структуру у вигляді кільцевого буфера (англ. ring buffer, vector dequeue), оскільки вона має попередньо виділене місце з країв масиву, що робить складність цих операцій амортизованою $O(1)$. При цьому операції вставлення будуть мати складність $O(n)$.

Аби зробити операції вставки більш ефективними, можна використовувати структуру на основі RRB векторів [14], але випадковий доступ сповільниться: у гіршому випадку індексація займатиме $O\left(log_2(m) \cdot log_m(n)\right)$; додавання у кінець займатиме $O\left(m \cdot log_m(n)\right)$; операції, які не використовуються у звичайному випадку, але які використовуються для реалізації вставки – розділення за $O\left(m \cdot log_m(n)\right)$ та конкатенація за $O\left(m^2 \cdot log_m(n)\right)$, де m – константа-фактор розгалуження (в ідеалі степінь двійки) наприклад, 32.

Для об’єктів має сенс використовувати звичайні хеш-таблиці, з функціями хешування та порівняння, оптимізованими для рядків.

Шлях до кожного поля у базі даних можна представити як: обов’язковий кореневий вузол у вигляді послідовності байтів, та шлях до дочірнього елементу, кожна ланка якого – це індекс або ключ (послідовність байтів).

3.4. Мережеве представлення даних

Для передачі по мережі краще за все обирати двійковий формат кодування. Це пришвидшить декодування та дозволить передавати у якості рядків довільний набір байтів.

Так, двійкове представлення шляху до значення можна визначити наступною РБНФ (англ. EBNF, extended Backus–Naur form) граматикою:

Як видно, ключі можуть представляти собою довільну послідовність байт. При цьому USIZE має представляти байти числа у порядку від старшого до молодшого (англ. big endian), а числове значення має бути менше ніж максимальне значення машинного слова операційної системи бази даних. У випадку 64-бітної системи, число має бути менше $2^{64} - 1$. Загальна кількість байт у представленні шляху не має перевищувати $\text{total length} - 8$.

Для представлення даних можливо використовувати формат на кшталт BSON.

ISSN 1560-8956

path = total length, root, subpath ;
root = root length, root length * BYTE ;
subpath = total parts, total parts * part ;
part = ‘0’, index
    | ‘1’, key ;
index = USIZE ;
key = key length, key length * BYTE ;
key length = USIZE ;
total length = USIZE ;
root length = USIZE ;
total parts = USIZE ;
USIZE = 8 * BYTE ;
BYTE = ‘0’ | ‘1’ | … | ‘255’ ;

Рисунок 2. Граматика двійкового представлення шляху

3.5. Поведінка команд СКБД

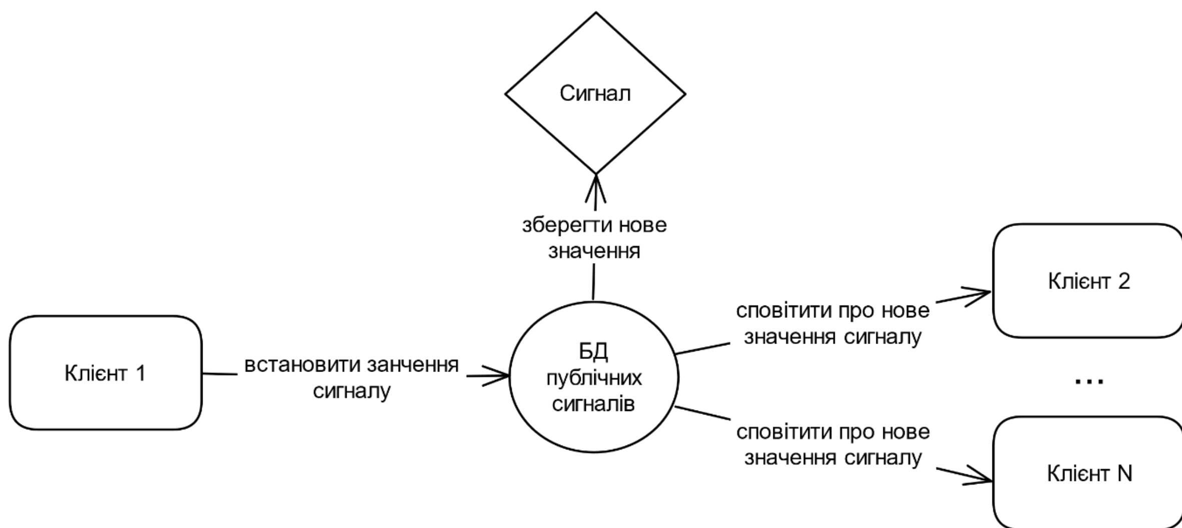

Рисунок 3. Схема розповсюдження оновлення значень сигналу.

Розглянемо поведінку СКБД при обробці команд.

Після отримання команди GET, система має розібрати шлях, за яким відбувається запит. Далі, система лінійно спускається по дереву об’єктів за $O(n)$, де n – довжина шляху та, в оптимістичному випадку, знаходить значення. Інакше, повідомляє клієнта про відсутність значення за даним шляхом. Якщо команда не мала значення параметру «не відстежувати» (англ. untracked) встановлене у істинне, то до відповідного значення, навіть якщо воно у базі даних відсутнє, додається асоційований підписник. У

подальшому, на стороні клієнтської бібліотеки, сигнал буде представлений у вигляді типу-суми з двома варіантами: значення відсутнє (англ. missing), значення присутнє (англ. present).

Після отримання команди SET, система має розібрати шлях, за яким відбувається встановлення значення, а також декодувати саме значення. У пригоді стає параметр total length шляху, що дозволяє пропустити потрібну кількість байтів. Таким чином зручно виконати паралелізацію розбору двійкового кодування шляху та значення, що дозволить трохи прискорити виконання. Розбір двійкового представлення значення також можливо підрозділити на паралельні потоки, якщо кодування буде надавати значення довжин фрагментів у байтах.

СКБД лінійно спускається по дереву об’єктів за $O(n)$, де n – довжина шляху та, в оптимістичному випадку, знаходить комірку, у яку потрібно встановити значення. Дана команда має наступні вимоги: комірка значення має бути зайнята, або бути полем об’єкту, або бути одразу перед початком/після кінця масиву. Інакше, клієнта буде повідомлено про помилку. У випадку, якщо умови були задоволені, усі підписники даного вузла дерева, а також його батьківських вузлів будуть повідомлені про нове значення вузла. Якщо вузол мав дочірні вузли, що були знищені в результаті зміни значення, їх підписники також про це повідомляються.

Команди, що викликають зміни значення не мають параметру untracked, оскільки його наявність може спричинити великі незручності під час розробки, у тому сенсі, що неможливо буде відслідкувати зміни значення, якщо певний клієнт про них не повідомляє. У деяких бібліотеках для реактивного програмування користувацьких інтерфейсів, (наприклад, Leptos) присутній функціонал, що дозволяє виконувати зміни сигналів без сповіщення про це його підписників. У локальному сценарії така поведінка не є критичною. Але у випадку, коли значення є доступним для будь-якої частини системи, це є неприпустимим, бо може спричинити поведінку та помилки, які дуже складно відлагодити.

Аналогічно до команди SET мають працювати й решта команд для зміни значень (PUSH BACK/FRONT, INSERT), але вони орієнтовані на роботу зі списками.

Команда DELETE приймає у аргументи лише шлях до вузла, що треба видалити. Операція є ідемпотентною. Якщо значення у базі даних відсутнє, то нічого не відбувається. Якщо значення присутнє – воно видаляється, відповідні поточні та дочірні підписники повідомляються про видалення, а батьківські – про зміну значення об’єкту/списку.

Аналогічним чином працюють команди POP BACK/FRONT, за винятком того, що застосовні лише до списків та повертають значення, що видалено, клієнту, у разі його наявності.

 ISSN 1560-8956

Розглянемо детально приклад послідовності подій при роботі СКБД, коли використовуються команди GET та SET (Рис. 4-5). Нехай «Клієнт 1» встановлює значення за шляхом, надіславши СКБД команду SET. СКБД записує значення в БД та надсилає відповідь із підтвердженням операції. СКБД перевіряє наявність підписників на цей шлях, але, оскільки їх не було, на цьому транзакція закінчується. Нехай після цього «Клієнт 2» запитає це значення у СКБД командою GET. СКБД паралельно виконує наступні дії: додає підписника за даним шляхом, та шукає й повертає значення за даним шляхом з бази даних. Якщо після цього який-небудь інший клієнт, наприклад «Клієнт 1», змінить значення за цим шляхом, то «Клієнт 2», як його підписник, отримає про це повідомлення та оновлене значення.

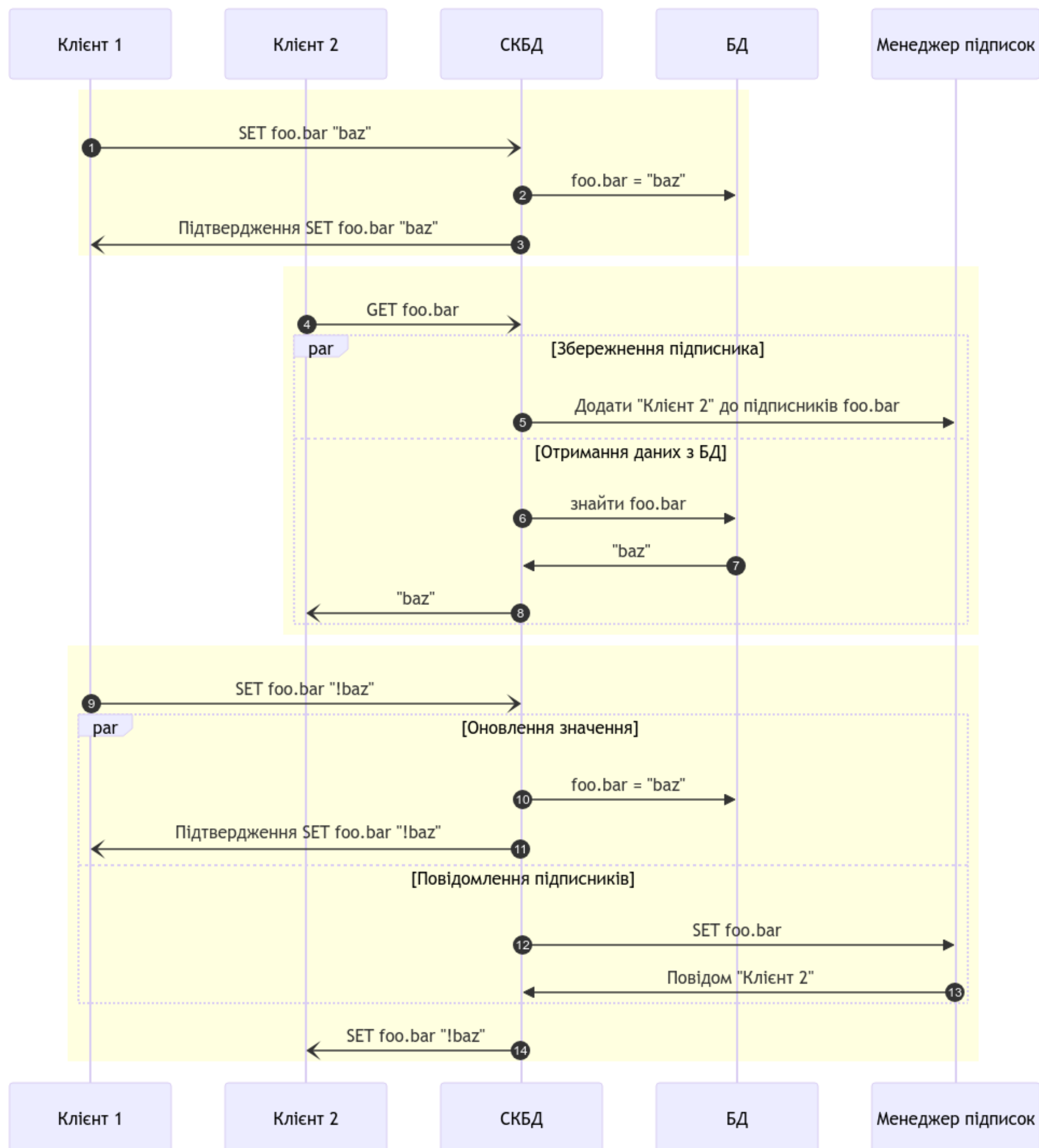

Рисунок 4. Схема послідовності взаємодії клієнтів та СКБД для команд GET/SET

Коли відбувається сповіщення підписників, воно має виконуватися після того як обидві паралельні операції завершилися, оскільки має відбуватися перевірка того, що нове значення відрізняється від попереднього. Самі сповіщення підписників не є напряму залежними між собою, тому можуть виконуватися паралельно. У даному випадку підписник один, тому паралельне сповіщення не спостерігається.

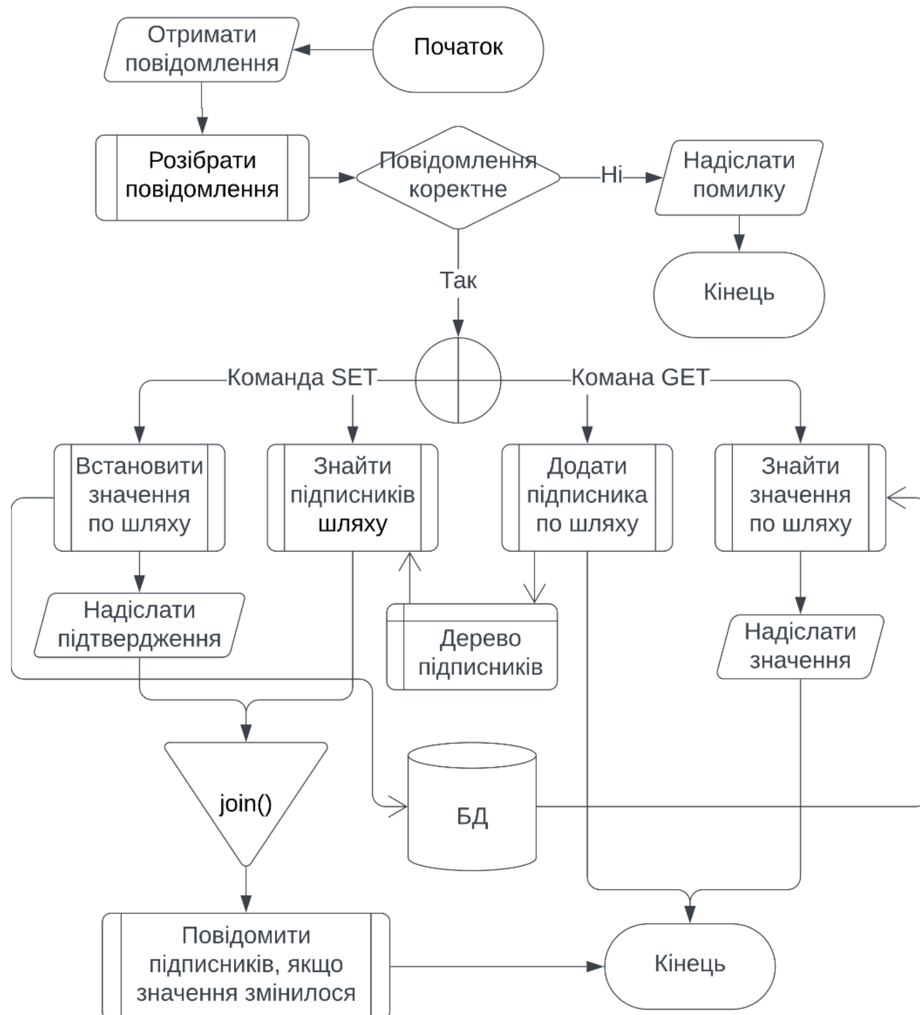

Рисунок 5. Високий рівень представлення алгоритму роботи команд СКДБ

3.6. Поведінка клієнтської бібліотеки

Розглянемо реалізацію клієнтської бібліотеки на основі реактивного фреймворку для веб-інтерфейсів «SolidJS» [15]. Даний фреймворк дозволяє описувати користувацький інтерфейс та керувати його поведінкою, використовуючи такі два основних види примітивів, як функціональні реактивні «сигнали» та «ефекти».

Після оголошення сигналу, користувач отримує два методи для роботи з його значенням: геттер та сеттер (англ. getter and setter). Ці методи дозволяють, відповідно, зчитувати та встановлювати значення сигналу.

 ISSN 1560-8956

Для отримання реактивної поведінки, сигнали використовують у комбінації з ефектами та обробниками подій. Ефект представляє собою частину коду, що виконує певний набір інструкцій, результат виконання яких може залежати від значень сигналів, які зчитуються геттерами. Зазвичай ефекти використовують для операцій виведення. Тим часом, для операцій введення стають у пригоді обробники подій, таких як, наприклад, натискання кнопки, або надсилання форми. Обробники подій можна використовувати для встановлення значень сигналів за допомогою сеттерів.

Після оголошення, ефект автоматично викликається один раз, під час чого підписується на сигнали, значення яких були зчитані. Якщо після цього значення сигналу буде змінене сеттером, усі підписані на нього ефекти будуть повторно викликані, й, можливо, підпишуться вже на інші сигнали, в залежності від потоку керування (англ. control flow).

Сигнали можливо компонувати у функції, що дозволяє робити код більш виразним.

Ці три шаблони (встановлення значень сигналів у обробниках подій під час операцій введення; використання значень сигналів у ефектах для операцій виведення; композиція сигналів у функції для виразності) дозволяють дуже зручно та декларативно будувати користувацький інтерфейс та логіку застосунку.

Користувач клієнтської бібліотеки для БД публічних сигналів спочатку має встановити зв’язок із СКБД (Рис. 6). Для зв’язку було використано протокол WebSocket. Даний протокол було обрано насамперед тому, що необхідно утримувати постійний двосторонній зв’язок між клієнтом та сервером, а цей протокол має реалізацію на багатьох мовах програмування. Для подальшої оптимізації швидкодії та комунікації, має сенс розробити власний протокол неперервного двостороннього зв’язку.

Після встановлення зв’язку стає можливим отримувати та надсилати повідомлення до СКБД. Усього передбачено 8 команд, представлених вище (розділ 3.5). Розглянемо реалізацію поведінки команд GET та SET як базових, на основі логіки яких можливо будувати поведінку решти (Рис. 7-9).

Тепер можемо перейти до висновків і сформулюємо, що вдалося зробити, і що ще належить зробити.

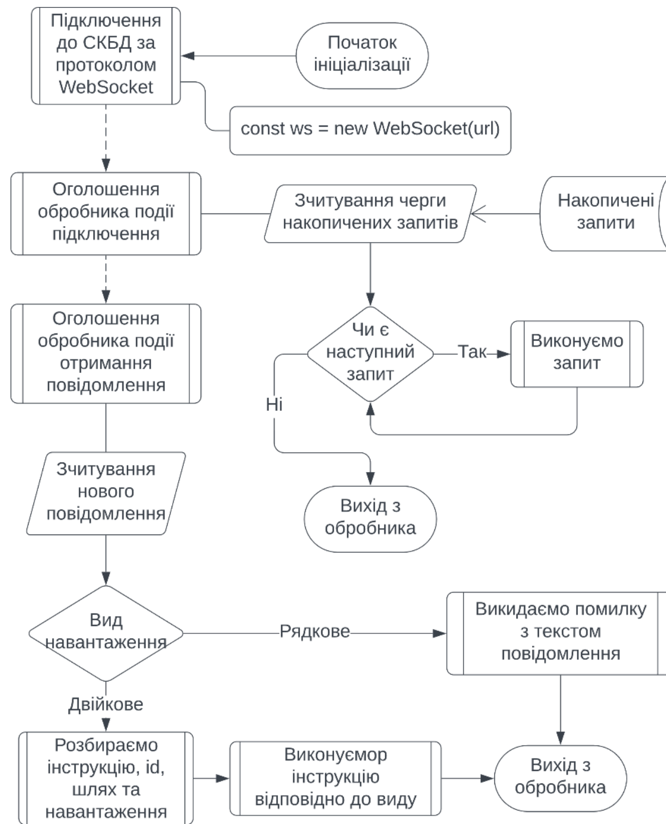

Рисунок 6. Алгоритм ініціалізації зв’язку

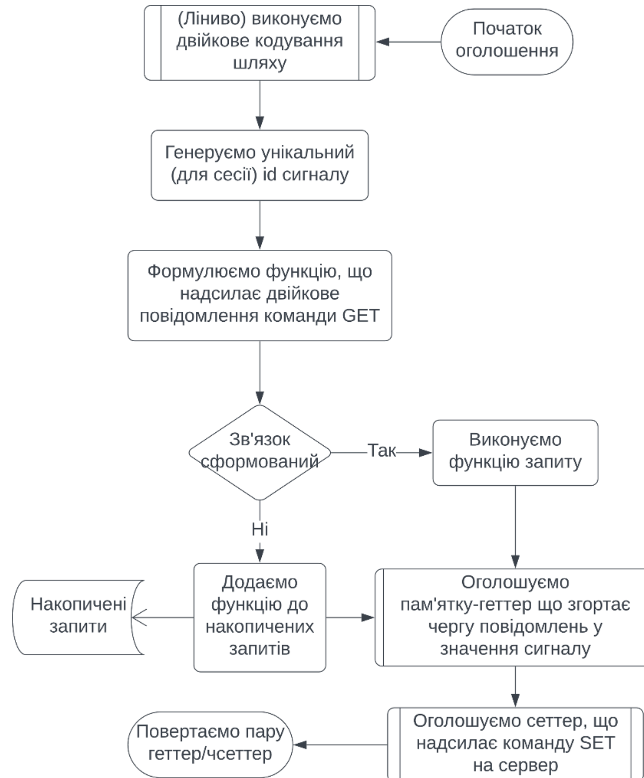

Рисунок 7. Алгоритм оголошення публічного сигналу (команда GET)

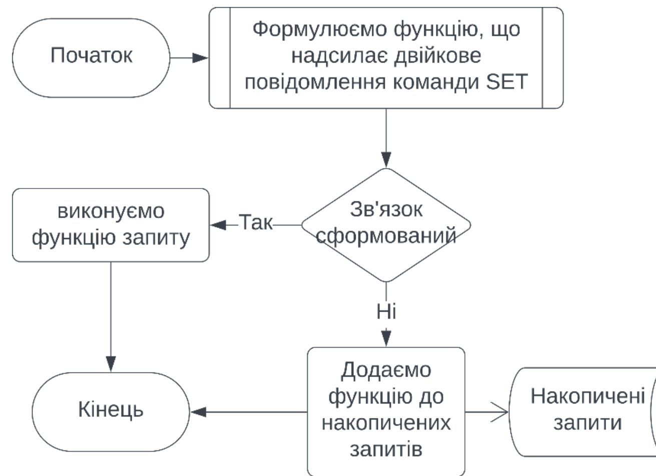

Рисунок 8. Алгоритм встановлення значення публічного сигналу (команда SET)

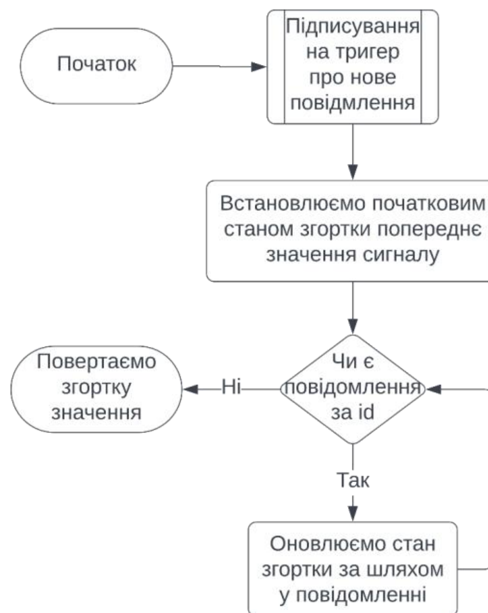

Рисунок 9. Згортка черги повідомлень від СКБД щодо значення сигналу

Висновки

Виконане дослідження проблеми підвищення ефективності комунікації між функціональними компонентами складних систем з мікросервісною архітектурою, та коректної синхронізації їх станів дозволяє зробити такі висновки.

На основі аналізу існуючих підходів до побудови застосунків із реактивним відстеженням змін у глобальних даних було запропоновано підхід до побудови бази даних «публічних сигналів», що автоматично сповіщає підписників про зміни у значеннях та їх дочірніх вузлах.

Було розглянуто можливий варіант двійкового представлення шляху до вузлів у БД, розібрано алгоритми поведінки СКБД та клієнтських бібліотек стосовно публікації та оновлення даних у БД.

У подальших дослідженнях та розробках має сенс сформувати власний двосторонній протокол зв’язку на заміну WebSocket, аби мінімізувати розмір пакетів та пришвидшити їх обробку. Варто розглянути стратегії керування пам’яттю зі сторони СКБД.

Для багатьох сценаріїв корисно мати статичну типізацію даних у БД, але наразі наявна лише динамічна типізація даних, що дає простір для існування більшої кількості помилок у поведінці застосунків-клієнтів. Тому, варто розробити способи встановлення статичної типізації даних. Має сенс додати більшу кількість примітивних типів даних.

Також, має сенс додати можливість зберігати у БД похідні сигнали та сигнали-тригери. Це дозволить розширити сценарії використання та варіанти міжсервісної комунікації. Для цього необхідно розробити інтерпретовану мову програмування, яка б дозволила виконувати комбінування сигналів у похідні.

Для розподіленої системи на основі даної СКБД необхідною також є реалізація стійких черг повідомлень для підписників, аби вони не втрачали послідовність змін у разі відмови клієнту або СКБД.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Моренець С.* Особливості event-driven architecture на прикладах з практики. *DOU.ua*. URL: https://dou.ua/forums/topic/38182/.
2. Redis Ltd. Redis. Version 7.2.5. 2024. URL: https://redis.io/.
3. Redis keyspace notifications. *Redis - The Real-time Data Platform*. URL: https://redis.io/docs/latest/develop/use/keyspace-notifications/.
4. Linux Foundation. Valkey. Version 7.2.5. 2024. URL: https://valkey.io/.
5. Supabase Inc. Supabase. Version 1.24.4. 2024. URL: https://supabase.com/.
6. Subscribing to Database Changes. *Supabase Docs*. URL: https://supabase.com/docs/guides/realtime/subscribing-to-database-changes.
7. MongoDB, Inc. MongoDB. Version 7.3.2. 2024. URL: https://www.mongodb.com/.
8. MongoDB, Inc. Change streams. *MongoDB Manual v7.0*. URL: https://www.mongodb.com/docs/manual/changeStreams/.
9. Broadcom Inc. RabbitMQ. Version 3.13.3. URL: https://www.rabbitmq.com/.
10. Broadcom Inc. RabbitMQ Documentation. *RabbitMQ*. URL: https://www.rabbitmq.com/docs.

 ISSN 1560-8956

11. *Kamina T., Aotani T.* An approach for persistent time-varying values. *SPLASH '19: 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Athens Greece. New York, NY, USA, 2019. URL: https://doi.org/10.1145/3359591.3359730.

12. *Kamina T., Aotani T.* Harmonizing Signals and Events with a Lightweight Extension to Java. *The Art, Science, and Engineering of Programming*. 2018. Vol. 2, no. 3. URL: https://doi.org/10.22152/programming-journal.org/2018/2/5.

13. *Kamina T., Ueno S.* Distributed Persistent Signals: Architecture and Implementation. *REBLS '22: 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, Auckland New Zealand. New York, NY, USA, 2022. URL: https://doi.org/10.1145/3563837.3568341.

14. RRB vector: a practical general purpose immutable sequence / N. Stucki et al. *ACM SIGPLAN Notices*. 2015. Vol. 50, no. 9. P. 342–354. URL: https://doi.org/10.1145/2858949.2784739.

15. *Carniato R. SolidJS.* Version 1.8.17. 2024. URL: https://www.solidjs.com/.