програмування, але й гарантувати створення надійного, стійкого до сучасних кіберзагроз та максимально зручного інтерфейсу, що відповідає найвищим стандартам якості програмного забезпечення.

**Список використаних джерел:**


1. React – A JavaScript library for building user interfaces. Official Documentation. URL: https://react.dev/.
2. MDN Web Docs: Front-end web developer. Mozilla Corporation. URL: https://developer.mozilla.org/en-US/docs/Learn/Front-end_web_developer
3. OWASP Top 10 Client-Side Security Risks. Open Worldwide Application Security Project. URL: https://owasp.org/www-project-top-10-client-side-security-risks/.
4. Web Security Guidelines. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/Security.
5. Osmani A. Learning JavaScript Design Patterns: A JavaScript and React Developer's Guide. 2nd Edition. O'Reilly Media, 2023. 280 p.



*УДК 004.41, 004.75*


## ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ КОНТРОЛЮ ВЕРСІЙ


*Базюк В.Р*
*makamagol@gmail.com*
*Черкаський державний фаховий бізнес-коледж*
*Марченко С. В.*
*Черкаси, Україна*


Розподілені системи контролю версій (Distributed Version Control Systems, DVCS) є критично важливим класом інженерного програмного забезпечення, що поєднує збереження історії змін, асинхронну реплікацію, цілісність даних і підтримку паралельної розробки. Сучасні системи цього класу, зокрема Git, Mercurial, Fossil і Darcs, реалізують подібні базові функції, але істотно

відрізняються за моделлю збереження об’єктів, організацією історії, механізмами синхронізації та оптимізаціями доступу до великих репозиторіїв. Оскільки саме ці внутрішні архітектурні рішення визначають масштабованість, швидкодію та надійність DVCS, аналіз їхніх функціональних компонентів і програмних реалізацій є актуальним інженерним завданням.

Здійснити критичний аналіз основних функціональних компонентів DVCS, дослідити варіації їх програмної реалізації в поширених системах контролю версій та визначити ключові архітектурні компроміси між продуктивністю, ефективністю збереження даних, зручністю синхронізації й складністю супроводу.

Git офіційно позиціонується як швидка, масштабована розподілена система керування версіями з доступом як до високорівневих, так і до внутрішніх механізмів. Порівняльна характеристика реалізацій DVCS наведена в табл. 1.

У сценаріях роботи з дуже великими репозиторіями (монорепозиторіями) кращі результати демонструє архітектура системи Git. Хороші результати досягаються поєднанням моделі збереження знімків стану (snapshot-model) із багаторівневою системою оптимізацій доступу до даних. Часткове клонування (partial clone) дає змогу завантажувати лише підмножину об’єктів, необхідних для поточного робочого контексту, а індекси досяжності об’єктів у вигляді бітових карт (bitmap indexes) і повторне використання пакетів об’єктів (pack reuse) суттєво скорочують час виконання операцій отримання змін (fetch) і клонування репозиторію (clone) [1; 7; 9].

Фактично це означає, що продуктивність Git забезпечується не лише базовою графовою моделлю історії у вигляді орієнтованого ациклічного графа (directed acyclic graph, DAG) і контентно-адресованим сховищем (content-addressable storage), а й використанням спеціалізованих індексних структур, які дозволяють замінити повний обхід історії попередньо обчисленими метаданими. Недоліком такого підходу є зростання складності супроводу репозиторію, оскільки для підтримання ефективності необхідно виконувати процедури перепакування (repack) об’єктів і збирання «сміття».

Таблиця 1 – Порівняльна характеристика реалізацій розподілених СКВ

| № | Функціональний компонент | Git | Mercurial | Fossil | Darcs |
|---|---|---|---|---|---|
| 1 | Модель збереження даних | Збереження знімків стану з подальшим пакуванням об’єктів (snapshot, packfiles) [1; 2] | Журнали ревізій із збереженням різниць між версіями (revlog, generaldelta) [3] | Збереження артефактів у вбудованій базі даних (SQLite) зі стисканням різниць [4] | Збереження змін як окремих патчів [5] |
| 2 | Модель історії змін | Орієнтований ациклічний граф комітів з додатковими індексами [6] | Граф змін (changeset DAG) [3] | Граф артефактів репозиторію (Artifact DAG) [4] | Алгебра змін із можливістю перестановки патчів (Patch algebra) [5] |
| 3 | Механізми реплікації | Інтелектуальний протокол синхронізації з вибірковим отриманням даних [7] | Передавання груп змін і бандлів історії використання [8] | Синхронізація через мережеві служби HTTP або SSH [4] | Обмін множинами патчів [5] |
| 4 | Алгоритми узгодження змін | Рекурсивне тристороннє злиття [1] | Тристороннє злиття (Three-way merge) [3] | Злиття на рівні файлів [4] | Формальна комутація змін (Patch commutation) [5] |
| 5 | Оптимізація продуктивності | Індекси досяжності, багатопакетні індекси, повторне використання пакетів [9] | Ланцюги різниць і оптимізоване стискання історії (Delta chains) [3] | Індексація на рівні бази даних [4] | Формалізоване впорядкування змін (Algebraic reasoning) [5] |

У випадках, коли історія розвитку проєкту характеризується значною кількістю розгалужень і частими операціями злиття, архітектура системи Mercurial може демонструвати кращі результати. Це пов’язано з використанням структури журналів ревізій (revlog) і формату загальних дельт (generaldelta), який дозволяє будувати ланцюги різниць (delta chains) між довільними версіями, а не лише між сусідніми [3]. Завдяки цьому досягається кращий коефіцієнт стискання історії змін і більш передбачувана продуктивність операцій запису нових змін. Модель append-only додавання зменшує необхідність модифікації вже збережених структур і тим самим підвищує надійність сховища. Водночас реконструкція повного стану репозиторію може вимагати обробки довших

послідовностей різниць, що збільшує затримку операції відновлення робочої директорії (checkout).

Для невеликих команд або автономних середовищ розробки ефективною може бути архітектура системи Fossil. Використання вбудованої системи керування базами даних SQLite як єдиного сховища дає змогу зберігати всі артефакти, метадані та індекси у вигляді одного файлу репозиторію [4]. Кращі результати тут досягаються за рахунок високого рівня інтеграції: система контролю версій, засоби відстеження помилок і вебінтерфейс адміністрування реалізовані в межах однієї програмної платформи. У результаті це зменшує залежність від зовнішньої інфраструктури та спрощує резервне копіювання. Разом із тим така централізація створює потенційні вузькі місця масштабування при інтенсивній паралельній роботі великої кількості розробників.

У дослідницьких сценаріях або спеціалізованих системах, де важливо формально описувати залежності між змінами, концептуальні переваги може мати архітектура системи Darcs. Вона базується на теорії патчів (patch theory), яка розглядає історію як множину змін, що можуть переставлятися за певних умов (commutation) [5]. Кращі результати в цьому випадку пов’язані з можливістю більш строгого логічного узгодження змін і потенційним зменшенням кількості конфліктів. Проте на практиці це супроводжується зростанням алгоритмічної складності аналізу залежностей і зниженням продуктивності при роботі з великими історіями змін, що обмежує застосування такого підходу у промислових масштабах.

У роботі запропоновано архітектуру вебплатформи проведення онлайн-вікторин у режимі реального часу. Її побудовано як modular monolith із чітким відокремленням REST-взаємодії, каналу передавання подій у режимі реального часу та модуля керування станом гри. Критичний аналіз показав, що для системи такого класу обраний підхід є обґрунтованішим за мікросервісну декомпозицію на ранньому етапі, оскільки дозволяє зменшити латентність, уникнути надмірної інфраструктурної складності та забезпечити узгоджений стан ігрової сесії.

**Список використаних джерел:**

1. Chacon S., Straub B. Pro Git. 2nd ed. New York : Apress, 2014. 456 p.
2. Git internals – Git objects . URL: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects (дата звернення: 16.03.2026).
3. O'Sullivan B. Mercurial: the definitive guide . URL: https://book.mercurial-scm.org/read/concepts.html (дата звернення: 16.03.2026).
4. Fossil SCM technical overview. URL: https://fossil-scm.org/home/doc/trunk/www/tech_overview.wiki (дата звернення: 16.03.2026).
5. Darcs theory . URL: https://darcs.net/Theory (дата звернення: 16.03.2026).
6. Git commit-graph documentation . URL: https://git-scm.com/docs/commit-graph (дата звернення: 16.03.2026).
7. Git partial clone . URL: https://git-scm.com/docs/partial-clone (дата звернення: 16.03.2026).
8. Mercurial wire protocol. URL: https://www.mercurial-scm.org/wiki/WireProtocol (дата звернення: 16.03.2026).
9. Git pack objects . URL: https://git-scm.com/docs/git-pack-objects (дата звернення: 16.03.2026).

*УДК 004.4:004.2*

# РЕТРО-ТЕХНОЛОГІЇ: ЧИ МОЖЛИВО ВІДТВОРИТИ ПРОГРАМУВАННЯ ДЛЯ ЗАСТАРІЛИХ ПЛАТФОРМ

*Собчук Є. О.*
*yehor.sobchuk@gmail.com*
*Черкаський державний фаховий бізнес-коледж*
*Люта М. В.*
*м. Черкаси, Україна*

Актуальність роботи зумовлена широким використанням застарілих інформаційних технологій у критично важливих галузях, таких як фінансовий сектор, державне управління, енергетика та охорона здоров'я. Незважаючи на стрімкий розвиток сучасних IT-рішень, значна частина інформаційних систем