УДК 004.658
DOI https://doi.org/10.32782/2663-5941/2024.3.1/07

Белоус Р.В.
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Крилов Є.В.
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

ОПТИМІЗАЦІЯ ВИКОРИСТАННЯ RAFT В РОЗПОДІЛЕНИХ СИСТЕМАХ З БАЗАМИ ДАНИХ

Констатовано, що оптимізація використання алгоритму RAFT (Réplicated State Machine) в розподілених системах з базами даних є важливим завданням для забезпечення стабільності та надійності системи. Наголошено, що метод Raft ґрунтується на використанні реплікованих машин стану, при цьому суть цього підходу полягає в тому, що конкретна група серверів має однаковий стан і може взаємозамінювати один одного у випадку виникнення проблем. У консенсусному алгоритмі Raft роль кожного учасника (вузла) в системі розділяється на три ключові категорії: Лідер (leader), що стежать (followers) і кандидати (candidates). Цей поділ на ролі є фундаментом, на якому будується вся система Raft. Лідер – це активний вузол, який тимчасово керує системою та приймає рішення щодо операцій, таких як запис даних. Він також відповідальний за ініціювання реплікації даних на інші вузли (стежать). Спостерігаючі вузли – це ті учасники системи, які стежать за лідером і виконують його команди. Вони отримують реплікаційні дані від лідера і зобов'язані дотримуватися узгодженості даних. Кандидати – це вузли, які бажають стати лідером. Вони ініціюють вибори, пропонуючи себе як нового лідера.

Алгоритм Raft використовує модель взаємодії за допомогою віддалених викликів (RPC), при цьому використовуючи лише дві віддалені процедури – RequestVote та AppendEntries. Значущою перевагою цього алгоритму є його здатність до автоматизованого управління кластером, що на практиці дозволяє вносити зміни в конфігурацію кластера без його зупинки. Ще одним важливим аспектом при використанні алгоритму є видалення застарілих запитів із журналу.

Перелічено та охарактеризовано такі методи оптимізації використання Raft в розподілених системах з базами даних, як налаштування параметрів алгоритму, реалізація асинхронного вводу-виводу, кешування стану, розпаралелювання операцій, реплікація даних, використання технік компресії, оптимізація мережевого взаємодії, поділ функціональності та розбиття на підкластери. Приведемо програмну реалізацію головних підходів до оптимізації досліджуваного алгоритму.

Ключові слова: *Кластеризація, Réplicated State Machine, Лідер, відмовостійкість, Кандидат, розпаралелювання.*

Постановка проблеми. Алгоритми консенсусу представляють собою ефективний механізм для забезпечення відмовостійкості, надаючи засіб автоматичного відновлення лідера. У системах з архітектурою сервер-клієнт сервер виступає як єдина точка відмови, тому важливо забезпечити наявність реплік лідера, що можуть прийняти його функції у випадку відмови. Алгоритми консенсусу, зокрема Raft, використовуються для забезпечення узгодженості стану між лідером та його репліками в кластері. Оптимізація використання алгоритму RAFT (Réplicated State Machine) в розподілених системах з базами даних стає важливим завданням для забезпечення стабільності та надійності системи [1].

Серед різних алгоритмів консенсусу, Raft є найбільш оптимальним для побудови систем, що складаються з основного сервера та його реплік. Оскільки архітектура розподіленої системи передбачає наявність основного сервера та його реплік для підвищення надійності, вибір застосування алгоритму Raft для оптимізації є належним рішенням.

Аналіз останніх досліджень і публікацій. Значні за обсягом дослідження із вивчення шляхів оптимізації використання Raft в розподілених системах з базами даних Хуанга Д. (Huang D.) [2], Паріса Ж. (Pâris J.) [4], Мельника Д. (Melnyk D.) [3].

Постановка завдання. Мета статті полягає в аналізі шляхів оптимізації використання Raft в розподілених системах з базами даних.

Методологічну основу дослідження становить сукупність методів, способів, підходів і прийомів наукового аналізу, зокрема, методів індукції

і дедукції, аналізу та синтезу, єдності історичного і логічного, абстракції, узагальнення, системного підходу до явищ, що вивчаються, а також порівняння і статистичні методи.

Виклад основного матеріалу. Метод Raft ґрунтується на використанні реплікованих машин стану. Суть цього підходу полягає в тому, що конкретна група серверів має однаковий стан і може замінити один одного у випадку неполадок. Наявність репліки стану є ключовою для забезпечення безпеки Raft, оскільки алгоритм гарантує, що якщо сервер вже зафіксував запис під певним індексом у журналі, інший сервер не може його перезаписати за тим самим індексом.

У Raft послідовники виступають як пасивні спостерігачі, обробляючи лише службові записи від лідера та кандидатів. Обробка запитів від клієнтів відбувається лідером, навіть якщо вони надходять через послідовників. Надійність алгоритму підтверджується його здатністю виявляти сервери з застарілими даними, оскільки кожен 50-ий службовий запит має термін, що дозволяє визначити актуальність інформації. Термін визначає одиницю часу для алгоритму, вказуючи на період лідерства [4].

Алгоритм Raft використовує взаємодію за допомогою віддалених викликів (RPC), використовуючи дві основні процедури – RequestVote та AppendEntries. RequestVote використовується для виборів кандидатів, в той час, як AppendEntries має більш універсальне застосування, синхронізуючи журнали послідовників з журналом лідера і передаючи періодичний сигнал про активність лідера. Вибори ініціюються в тому випадку, якщо послідовники перестають отримувати сигнал від лідера або отримують його пізніше, ніж передбачено їх конфігурацією.

У Raft, кожен послідовник підтримує одного кандидата, і вибори можуть завершитися обранням лідера або залишитися безрезультатними. У випадку, коли сервери висувають свої кандидатури майже одночасно і голоси розподіляються між ними, можливий сценарій, коли ніхто не отримує більшість голосів. Raft вирішує цю проблему, використовуючи різні таймаути для виборів у різних послідовників [2].

Таким чином, лише один сервер може першим висунути свою кандидатуру і отримати більшість голосів після таймауту. Навіть із вимогою до алгоритму, яка стверджує, що несвоєчасні дії не повинні впливати на безпеку використання Raft, вчасний вибір лідера є критичним під час виборів.

Реплікація журналу є ще однією важливою частиною алгоритму. Кожен клієнт надсилає команду для виконання машиною стану. Лідер записує команду в журнал і пересилає цей запис послідовникам, щоб вони теж його зафіксували у своїх журналах. Після цього команда виконується, і результат передається клієнту. Кожна команда в журналі має індекс та термін, і лише фіналізовані команди виконані та повернуті клієнту. Лідер зберігає індекс останньої фіналізованої команди та передає цю інформацію в запиті AppendEntries, щоб послідовники також отримували дані про виконані команди [3].

Для перевірки актуальності журналу послідовника, лідер включає індекс і термін передостанньої команди в службовий запит. У випадку відповіді послідовника помилкою вказується, що його журнал застарілий. В такому випадку лідер вирішує розбіжності, замінюючи записи послідовника своїми за допомогою процедури AppendEntries. Лідер продовжує відправляти AppendEntries із старішими командами до успішної відповіді послідовника. Важливою перевагою Raft є його автоматизоване управління кластером, що дозволяє вносити зміни в конфігурацію кластера без його зупинки, за допомогою концепції спільного консенсусу між серверами із старою та новою конфігурацією.

У першій фазі застосування нової конфігурації сервери з обох конфігурацій беруть участь у виборах та записують однакові записи до своїх журналів. Лідер поступово розсилає всім послідовникам нову конфігурацію, яку вони починають застосовувати.

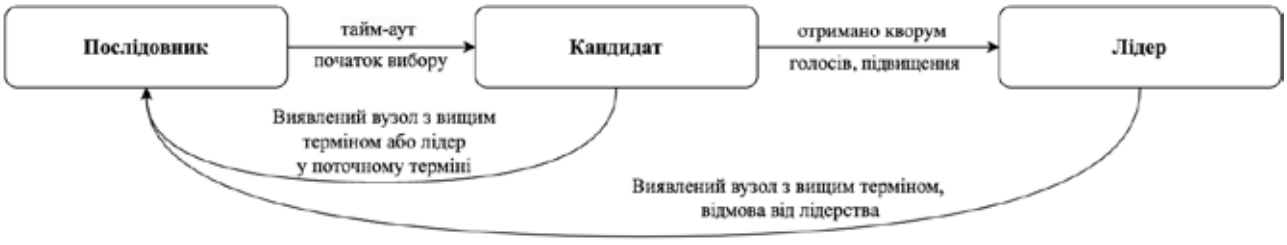

Рис. 1. Схематичне зображення алгоритму Raft

Ще одним важливим аспектом при застосуванні алгоритму є очищення журналу від застарілих запитів. З урахуванням того, що з більшим потоком запитів від клієнтів розмір журналу збільшується, необхідні механізми для його зменшення. Одним зі способів зменшення журналу є створення знімків поточного журналу та збереження його в окремому сховищі даних.

Для всіх налаштувань використовуються файли в форматі json – config. json. Основні шляхи оптимізації використання алгоритму RAFT в розподілених системах з базами даних включає:

– Налаштування параметрів алгоритму RAFT: Оптимальне налаштування тайм-аутів інтервалів вибору лідера (election timeouts) та битів (heartbeat intervals) дозволяє забезпечити швидше виявлення втрати лідерства та підтримувати актуальність копій даних.

– Реалізація асинхронного вводу-виводу: Використання асинхронного вводу-виводу може покращити продуктивність системи, дозволяючи іншим операціям чекати завершення вводу-виводу без блокування виконання.

– Кешування стану: Зберігання копій стану у кеші дозволяє швидше відновлювати стан системи після відновлення збоїв або перезапуску вузла.

– Розпаралелювання операцій: Розпаралелювання операцій обробки логів та інших обчислень може покращити продуктивність кожного вузла та загальну продуктивність системи.

– Реплікація даних: Збільшення кількості реплік бази даних дозволяє підвищити надійність системи та прискорити читання, оскільки можна читати дані з найближчого вузла.

– Використання технік компресії: Застосування методів компресії даних може зменшити обсяг передаваних повідомлень між вузлами, що дозволяє скоротити час передачі та зменшити обсяг мережевого трафіку.

– Моніторинг та логування: Ефективне використання моніторингу та логування дозволяє вчасно виявляти проблеми та відслідковувати дії системи для подальшого аналізу та оптимізації.

– Оптимізація мережевого взаємодії: Мінімізація мережевого трафіку та використання оптимальних мережевих налаштувань сприяє швидшому обміну даними між вузлами.

– Поділ функціональності: Розподіл функціональності між різними вузлами архітектури дозволяє розпаралелювати обробку запитів та зменшити навантаження на кожен конкретний вузол.

– Шкальованість: Розробка системи з урахуванням можливості горизонтального та вертикального шкалювання дозволяє легше адаптувати систему до зростання обсягу даних та трафіку.

– Розбиття на підкластери: якщо Raft-система стикається з великим навантаженням, то доцільним є розбиття її на кілька підкластерів. Кожен підкластер буде працювати як окрема система з власними лідером і стежать учасниками. Приклад коду (на псевдокоді) для управління кластеризацією:

– *class ClusterManager:*
– *def __init__(self):*
– *self. clusters = []*
– *def createCluster (self, clusterId, nodes):*
– *cluster = RaftCluster (clusterId, nodes)*
– *self. clusters. append (cluster)*
– *def getCluster (self, clusterId):*
– *for cluster in self. clusters:*
– *if cluster. clusterId == clusterId:*
– *return cluster*
– *class RaftCluster:*
– *def __init__(self, clusterId, nodes):*
– *self. clusterId = clusterId*
– *self. nodes = nodes*
– *self. leader = None*

Врахування цих аспектів та їхнє належне впровадження допомагає оптимізувати використання алгоритму raft у розподілених системах з базами даних для досягнення оптимальної продуктивності та надійності.

Проаналізуємо черговість дій для практичного виконання оптимізації використання Raft в розподілених системах з базами даних. У консенсусному алгоритмі Raft роль кожного учасника (вузла) в системі розділяється на три ключові категорії: Лідер (leader), що стежать (followers) і кандидати (candidates). Цей поділ на ролі є фундаментом, на якому будується вся система Raft. Наведемо приклад коду (на псевдокоді) для визначення ролі вузла:

```
class Node:
def __init__(self, id):
self. id = id
self. state = “follower” # Починаємо як наступний
self. currentTerm = 0
self. votedFor = None
def becomeCandidate (self):
self. state = “candidate”
self. currentTerm += 1
self. votedFor = self. id
def becomeFollower (self, term, votedFor):
self. state = “follower”
self. currentTerm = term
self. votedFor = votedFor
def becomeLeader (self):
self. state = “leader”
```

У цьому фрагменті коду ми створюємо клас Node, що представляє вузол у системі Raft, і визначаємо методи для зміни його стану. Коли вузол хоче стати кандидатом, він викликає becomeCandidate (), а для переходу в режим слідчого – becomeFollower(). Існує також метод becomeLeader (), який дозволяє вузлу стати лідером.

Тайм-ути і вибори лідера – важлива частина роботи алгоритму Raft. Якщо система виявить, що поточний лідер відсутній або недоступний, вона ініціює вибори нового лідера.

Тайм-аут (timeout): Кожен вузол в системі Apt налаштований на випадковий тайм-аут, який визначає, як часто вузол перевіряє стан лідера. Якщо вузол не отримує повідомлення від лідера протягом заданого інтервалу часу, він починає процес виборів.

Процес вибору лідера: Під час виборів вузли конкурують за роль лідера. Кожен кандидат надсилає запити (прохання про голос) іншим вузлам. Якщо він отримує більшість голосів, то стає новим лідером.

Приклад коду для тайм-аутів та процесу вибору лідера:

```
class Node:
def __init__(self, id, electionTimeout):
self. id = id
self. state = “follower”
self. currentTerm = 0
self. votedFor = None
self. log = []
self. commitIndex = 0
self. electionTimeout = electionTimeout
# Метод для перевірки настання тайм-аута
def checkTimeout (self):
# Логіка перевірки тайм-ауту
pass
# Метод для ініціювання виборів
def startElection (self):
# Логіка виборів pass
```

У даному коді ми маємо параметр electionTimeout, який представляє випадковий тайм-аут для кожного вузла. Метод checkTimeout перевіряє, чи настав тайм-аут, і якщо це сталося, вузол починає процес виборів за допомогою методу startElection.

Написання повноцінного програмного коду для оптимізації використання алгоритму RAFT в розподілених системах з базами даних – завдання велике та залежить від конкретного контексту та вимог вашого проєкту. В даній статті наведемо приклад простого коду мовою програмування Python, що використовує бібліотеку raft, яка реалізує алгоритм RAFT.

```
from raft import RaftNode
class CustomStateMachine:
def apply_log_entry (self, entry):
# Логіка застосування запису до стану
def query_state (self):
# Запит на поточний стан
def main():
# Конфігурація вузла
node_id = 1
peers = [2, 3, 4] # інші вузли в системі
# Створення вузла RAFT
raft_node = RaftNode (node_id, peers, CustomStateMachine())
# Запуск вузла
raft_node. start()
# Ваша логіка роботи з базою даних та інші оптимізації
# Зупинка вузла при завершенні роботи
raft_node. stop()
if __name__ == "__main__":
main()
```

Цей код використовує просту реалізацію CustomStateMachine для застосування логів та використовує базовий клас RaftNode з бібліотеки raft.

Висновки. Проведено аналіз використання алгоритму RAFT (Réplicated State Machine) в розподілених системах з базами даних та наголошено на важливості даного завдання для забезпечення стабільності та надійності системи. Перелічено та охарактеризовано такі методи оптимізації використання Raft в розподілених системах з базами даних, як налаштування параметрів алгоритму, реалізація асинхронного вводу-виводу, кешування стану, розпаралелювання операцій, реплікація даних, використання технік компресії, оптимізація мережевого взаємодії, поділ функціональності та розбиття на підкластери. Приведемо програмну реалізацію головних підходів до оптимізації досліджуваного алгоритму.

Список літератури:

1. Howard H. ARC: Analysis of Raft Consensus. [Електронний ресурс]. *University of Cambridge, Computer Laboratory.* 2014. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR857.pdf.

2. Huang D. Performance Analysis of the Raft Consensus Algorithm for Private Blockchains [Електронний ресурс]. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. 2020. URL: https://ieeexplore.ieee.org/document/8666147.

3. Melnyk D. Improving Raft When There Are Failures [Електронний ресурс]. *Distributed Computing Group, ETH Zurich. 2018 Eighth Latin-American Symposium on Dependable Computing (LADC*). 2018. URL: https://ieeexplore.ieee.org/document/8671595.

4. Pâris J. Pirogue, a lighter dynamic version of the Raft distributed consensus algorithm [Електронний ресурс]. *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. 2015. URL: https://ieeexplore.ieee.org/document/7410281.

Belous R.V., Krylov Ye.V. OPTIMIZATION OF RAFT USAGE IN DISTRIBUTED SYSTEMS WITH DATABASES

It is noted that optimizing the usage of the RAFT (Réplicated State Machine) algorithm in distributed systems with databases is a crucial task for ensuring system stability and reliability. Emphasis is placed on the fact that the Raft method is based on the use of replicated state machines, where a specific group of servers has an identical state and can replace each other in case of issues. In the Raft consensus algorithm, the role of each participant (node) in the system is divided into three key categories: Leader, Followers, and Candidates. This division into roles forms the foundation upon which the entire Raft system is built. The Leader is an active node temporarily in control of the system and makes decisions regarding operations such as data recording. It is also responsible for initiating data replication to other nodes (Followers). Followers are participants who observe the Leader and execute its commands. They receive replication data from the Leader and are obligated to maintain data consistency. Candidates are nodes that wish to become the Leader and initiate elections by proposing themselves as the new Leader.

The Raft algorithm operates on a remote procedure call (RPC) interaction model, utilizing only two remote procedures: RequestVote and AppendEntries. A significant advantage of the algorithm is its automated cluster management, allowing changes to the cluster configuration without stopping it. Another crucial aspect of applying the algorithm is clearing the log of outdated requests.

Various optimization methods for Raft usage in distributed systems with databases are enumerated and characterized, including tuning algorithm parameters, implementing asynchronous input-output, caching state, parallelizing operations, data replication, using compression techniques, optimizing network interaction, functional partitioning, and clustering. The programmatic implementation of these optimization approaches is provided.

Key words: *Clustering, Réplicated State Machine, Leader, Fault Tolerance, Candidate, Parallelization.*