https://doi.org/10.25559/SITITO.020.202403.715-725
УДК 004.75

Оригинальная статья

К вопросу реализации алгоритма консенсуса Raft на основе механизма корутин и специализированной сетевой библиотеки

М. О. Мельников*, Е. В. Игонина
ФГБОУ ВО «Елецкий государственный университет им. И. А. Бунина», г. Елец, Российская Федерация
Адрес: 399770, Российская Федерация, Липецкая область, г. Елец, ул. Коммунаров, д. 28-1
* melnikov.maxx@yandex.ru

Аннотация
В данной статье рассматривается реализация модуля консенсуса Raft Server на языке программирования C++ стандарта 20. Ключевой особенностью работы является то, что алгоритм реализован без использования сторонних библиотек. Такой подход гарантирует гибкость и дает возможность сделать упор на обеспечение максимальной производительности. Вначале представлен обзор алгоритма Raft, затем подробно изложен процесс разработки Raft Server, а также представлено описание собственной сетевой библиотеки, основанной на механизме корутин. Реализация алгоритма использует возможности C++ стандарта 20, в частности корутины, для представления эффективной и современной методологии создания критических компонентов распределённых систем. В данном исследовании не только демонстрируется практическое применение и преимущества корутин C++20 в сложных распределённых программных средах, но и проводится анализ возникающих проблем и их решений при разработке модуля консенсуса, такого как Raft Server.

Ключевые слова: консенсус, консенсус-алгоритмы, raft, блокчейн

Конфликт интересов: авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Мельников М. О., Игонина Е. В. К вопросу реализации алгоритма консенсуса Raft на основе механизма корутин и специализированной сетевой библиотеки // Современные информационные технологии и ИТ-образование. 2024. Т. 20, № 3. С. 715-725. https://doi.org/10.25559/SITITO.020.202403.715-725

Original article

Implementation of the Raft Consensus Algorithm Based on the Coroutine Mechanism and a Specialized Network Library

M. O. Melnikov*, E. V. Igonina
Bunin Yelets State University, Yelets, Russian Federation
Address: 28-1 Kommunarov St., Yelets 399770, Lipetsk region, Russian Federation
* melnikov.maxx@yandex.ru

Abstract
This paper discusses the implementation of the Raft Server consensus module in the C++ standard 20 programming language. The key feature of the work is that the algorithm is implemented without the use of third-party libraries. This approach guarantees flexibility and provides the opportunity to emphasize on maximizing performance. First, an overview of the Raft algorithm is presented, followed by details of the Raft Server development process, and a description of the proprietary network library based on the coroutine mechanism is presented. The implementation of the algorithm utilizes the features of C++ standard 20, in particular coroutines, to present an efficient and modern methodology for creating critical components of distributed systems. This study not only demonstrates the practical applications and benefits of C++20 coroutines in complex distributed software environments, but also analyzes the emerging problems and their solutions when developing a consensus module such as Raft Server.

Keywords: consensus, consensus algorithms, raft, blockchain

Conflict of interests: The authors declares no conflict of interest.

For citation: Melnikov M.O., Igonina E.V. Implementation of the Raft Consensus Algorithm Based on the Coroutine Mechanism and a Specialized Network Library. *Modern Information Technologies and IT-Education.* 2024;20(3):715-725. https://doi.org/10.25559/SITITO.020.202403.715-725

Введение

Представим что наша цель – разработать распределённую систему для хранения ключ-значений (K/V). В языке программирования C++ такая задача может быть легко решена с использованием контейнера `unordered_map<string, string>`. Однако в реальных приложениях требования к отказоустойчивости системы хранения значительно усложняют задачу. Простым решением может быть развёртывание трех (или более) нод, каждая из которых будет содержать реплику этого сервиса. Ожидается, что репликация данных и поддержка их согласованности пойдёт на откуп пользователей. Однако такой подход может привести к непредсказуемому поведению, например, возможна ситуация, когда данные по ключу обновляются, а затем при запросе возвращается их более старая версия.

На практике пользователи хотят, чтобы распределённая система, состоящая из нескольких узлов, функционировала так же эффективно, как и система, работающая на одном узле [1, 2]. Для выполнения этого требования перед системой хранения (или любым другим подобным сервисом, (далее «машина состояния») обычно размещается модуль консенсуса [3, 4]. Этот модуль гарантирует, что все взаимодействия пользователей с машиной состояния проходят исключительно через него, а не напрямую. Поэтому рассмотрим, как реализовать подобный модуль консенсуса на основе алгоритма Raft.

Материалы и методы

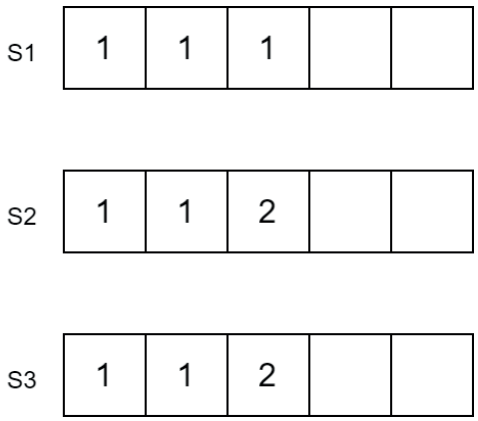
Для решения вопросов, связанных с реализацией модуля консенсуса Raft на языке программирования C++ стандарта 20, представлен обзор алгоритма Raft и его ключевых принципов. Далее подробно рассматривается процесс разработки самого модуля, акцентируя внимание на особенностях реализации, включая создание собственной сетевой библиотеки на основе корутин. Описывается подход, при котором алгоритм реализован без сторонних библиотек, что обеспечивает гибкость и позволяет оптимизировать производительность системы. В статье исследуется применение возможностей C++20, включая корутины, для построения современных и эффективных компонентов распределённых систем. Кроме того, проводится анализ возникающих проблем и предлагаются решения, которые были использованы при реализации Raft Server. В заключении даются рекомендации по использованию C++20 для разработки критически важных компонентов распределённых систем и определяются перспективы дальнейшей работы в этой области.

1. Алгоритм Raft

В алгоритме Raft может участвовать какое угодно количество узлов, называемых «пирами» *(peers)* [5-7]. Каждый узел ведет собственный журнал записей. Среди пиаров выделяется один «лидер» *(leader)*, остальные действуют в роли «последователей» *(followers)*. Все запросы пользователей (чтение и запись) направляются к лидеру. При получении запроса на изменение состояния машины лидер сначала записывает его в журнал, а затем пересылает последователям, которые также добавляют его в свои журналы [8, 9]. После того как большинство пиров успешно подтвердит запись, лидер считает запись зафиксированной, применяет её к машине состояния и уведомляет пользователя об успешном выполнении операции.

Терм *(Term)* является ключевым понятием в Raft и может только увеличиваться. Терм изменяется при системных изменениях, таких как, например, смена лидера. Журнал в Raft имеет специфическую структуру, каждая запись в нём состоит из Терма и Данных *(Payload)*. Терм указывает на лидера, который создал данную запись, а Данные представляют собой изменения, которые необходимо внести в машину состояния. Raft гарантирует, что две записи с одинаковым индексом и термом идентичны.

Журналы в Raft не являются строго добавочными *(append only)* и могут быть усечены. Например, в сценарии ниже лидер S1 реплицировал две записи перед тем, как выйти из строя [10, 11]. S2 стал новым лидером и начал репликацию записей, в результате чего журналы S1 стали отличаться от журналов S2 и S3. В этом случае последняя запись в журнале S1 будет удалена и заменена новой (рис. 1).



Р и с. 1. Сценарий с репликацией двух записей
F i g. 1. Scenario with replication of two records

Источник: здесь и далее в статье все таблицы и рисунки составлены авторами.
Source: Hereinafter in this article all tables and figures were made by the authors.

2. Raft RPC API

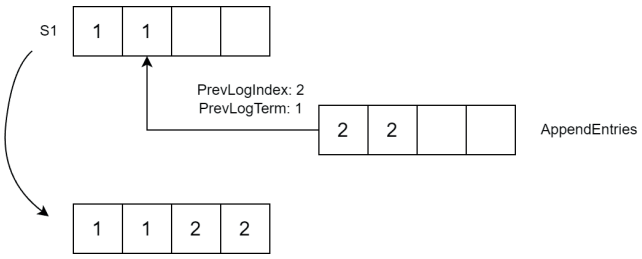
Рассмотрим RPC интерфейс Raft. Стоит отметить, что API *(application programming interface)* Raft включает всего два вызова. Raft гарантирует наличие только одного лидера на каждый терм. Также возможны термы без лидера, например, если выборы завершились неудачей [12, 13]. Чтобы гарантировать, что проводится только одни выборы, каждый узел сохраняет свой голос в постоянной переменной под названием `VotedFor`. RPC *(remote process call)* для выборов называется `RequestVote` и принимает три параметра: `Term, LastLogIndex` и `LastLogTerm`. Ответ содержит `Term` и `VoteGranted`. Примечательно, что каждый запрос содержит nthv, и Raft узлы могут эффективно взаимодействовать только если их термы совместимы.

Когда узел инициирует выборы, он отправляет запрос `RequestVote` другим узлам и собирает их голоса. Если большинство ответов положительные, узел переходит в статус лидера.

Теперь рассмотрим запрос `AppendEntries`. Он принимает такие

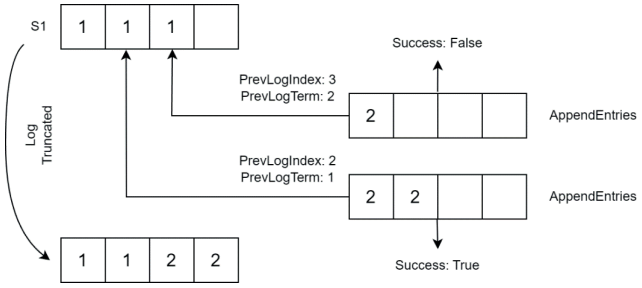
параметры, как `Term`, `PrevLogIndex`, `PrevLogTerm` и `Entries`, а ответ содержит `Term` и `Success`. Если поле `Entries` в запросе пустое, оно выполняет роль `Heartbeat` (сигнала о «живучести» узла).

Когда узел-последователь получает запрос `AppendEntries`, он проверяет `PrevLogIndex` на соответствие термину (рис. 2). Если он совпадает с `PrevLogTerm`, последователь добавляет записи (`Entries`) в журнал, начиная с `PrevLogIndex + 1` (при этом существующие записи после `PrevLogIndex` удаляются, если они есть).



Р и с. 2. Схема запроса AppendEntries
F i g. 2. AppendEntries Query Schema

Если термины не совпадают, последователь возвращает ‘`Success=false`‘. В этом случае лидер повторяет отправку запроса, уменьшая значение ‘`PrevLogIndex`‘ на единицу (рис. 3).



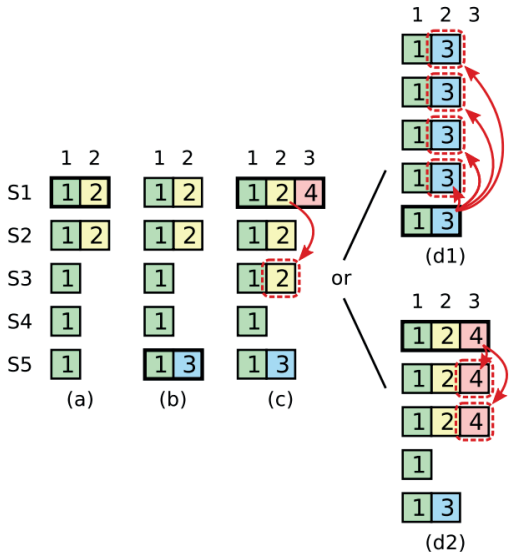
Р и с. 3. Успешное и ошибочное завершение запроса
F i g. 3. Successful and Error Completion of a Request

Когда узел получает запрос `RequestVote`, он сравнивает свои пары `LastTerm` и `LastLogIndex` с самой последней записью в журнале. Если эти пары менее или равны по значению парам отправителя запроса, узел возвращает `VoteGranted=true`.

3. Переходы между состояниями в Raft

Каждый узел изначально находится в состоянии Follower. Когда узел в состоянии Follower не получает запросов в течение заданного тайм-аута, он увеличивает значение параметра Term и переходит в состояние Candidate, инициируя процесс выборов. В случае успешного завершения выборов узел переходит в состояние Leader, в противном случае возвращается в состояние Follower. При получении RPC-запроса с параметром `Term`, превышающим его текущий, узел, независимо от своего состояния, переключается в состояние Follower [14, 15].

Узел S1 был лидером в `Term 2`, где он реплицировал две записи перед тем, как выйти из строя. Затем лидерство в `Term 3` перешло к узлу S5, который добавил запись и также вышел из строя. Далее узел S2 взял на себя лидерство в `Term 4`, реплицировал запись из `Term 2`, добавил свою собственную запись для `Term 4` и также вышел из строя. Это приводит к двум возможным исходам: либо узел S5 восстанавливает лидерство и усекает записи из `Term 2`, либо узел S1 вновь становится лидером и фиксирует записи из `Term 2` [16]. Записи из `Term 2` считаются надежно зафиксированными только после того, как они будут покрыты последующей записью от нового лидера. Этот пример был взят из диссертации Диего Онгаро[1] и представлен на рисунке 4.



Р и с. 4. Схема коммитов в Raft[2]
F i g. 4. Raft Commit Scheme

Пример иллюстрирует, как алгоритм Raft функционирует в динамичных и зачастую непредсказуемых условиях [17, 18]. Последовательность событий, включающая несколько лидеров и сбои, демонстрирует сложность поддержания согласованного состояния в распределенной системе. Эта сложность не является очевидной на первый взгляд, но становится значимой в ситуациях, связанных с изменением лидера и сбоями системы [19-21]. Пример подчеркивает важность надежного и продуманного подхода к управлению такими сложностями, что и является основной целью алгоритма Raft.

4. Реализации сервера Raft

Перейдем к реализации сервера Raft. Значительные преимущества для реализации предоставляет использование корутин C++20. В нашей реализации персистентное состояние хранится в оперативной памяти (RAM). Однако в реальных

[1] Ongaro D. Consensus: Bridging Theory and Practice : Dissertation for the Degree of Doctor of Philosophy. CA, USA: Stanford University, 2014. 240 p. URL: https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf (дата обращения: 12.05.2024).

[2] Там же. С. 23.

условиях его следует сохранять на диск [22, 23]. Далее мы подробнее поговорим о MessageHolder. Он работает аналогично shared_ptr из стандартной реализации C++, но специально предназначен для обработки сообщений Raft, обеспечивая эффективное управление и обработку этих коммуникаций.

```
struct TState {
    std::vector<TMsgHolder<TLogEntry>> log;
   uint32_t voted_for=0;
   uint32_t current_term=1;
};
```

В Нестабильном Состоянии *(volatile state)* обозначили записи как L для «лидера» или F для «последователя», чтобы уточнить их назначение. CommitIndex указывает на последнюю запись журнала, которая была зафиксирована. В отличие от этого, LastApplied обозначает наиболее недавнюю запись журнала, примененную к машине состояний, и всегда меньше или равен CommitIndex. NextIndex важен, поскольку он указывает на следующую запись журнала, которую необходимо отправить узлу-партнеру [24, 25]. Аналогичным образом, MatchIndex отслеживает последнюю запись журнала, которая обнаружила совпадение. Раздел Votes содержит идентификаторы узлов, проголосовавших за меня. Таймауты являются важным аспектом управления: HeartbeatDue и RpcDue управляют таймаутами лидера, в то время как ElectionDue отвечает за таймауты последователей.

```
using time_point_t = std::chrono::time_point<std::chrono::steady_clock>;
struct volatile_state_t {
    time_point_t election_due_time;
    std::unordered_set<int32_t> received_votes;
    std::unordered_map<int32_t, uint64_t> next_log_index;
      std::unordered_map<int32_t, uint64_t> matched_log_index;
     std::unordered_map<int32_t, time_point_t> heartbeat_expiry;
    std::unordered_map<int32_t, time_point_t> rpc_expiry;
    int32_t last_log_applied = 0;
    int32_t commit_log_index = 0;
};
```

5. Raft API

В нашей реализации алгоритма Raft используются два класса. Первый класс – INode, который представляет узел. Этот класс включает два метода: Send, который сохраняет исходящие сообщения во внутреннем буфере, и Drain, который обрабатывает фактическую отправку сообщений. Второй класс – Raft, который управляет состоянием текущего узла. Он также включает два метода: Process, который обрабатывает входящие соединения, и ProcessTimeout, который должен вызываться регулярно для управления таймаутами, такими как таймаут выборов лидера. Пользователи этих классов должны использовать методы Process, ProcessTimeout и Drain по мере необходимости. Метод Send класса INode вызывается внутренне в классе Raft, что обеспечивает бесшовную интеграцию обработки сообщений и управления состоянием в рамках алгоритма Raft.

```
struct INode {
   void Drain();
   void Send(TMsgHolder<TMessage> message);
   ~INode();
};
class TRaft {
public:
    TRaft(int32_t one_node, unordered_map<int32_t, shared_ptr<INode>>& nodes);
    void ProcessTimeout(Time today);
   void Process(Time now, TMsgHolder<TMessage> msg, shared_ptr<INode>& reply_to);
};
```

Теперь рассмотрим, реализована отправка и чтение сообщений Raft. Вместо использования библиотеки сериализации мы читаем и отправляем сырые структуры в формате TLV *(Type-Length-Value)*. Вот как выглядит заголовок сообщения:

```
struct TMessage {
    char Value[0];
    int32_t Len;
   int32_t Type;
};
```

Для дополнительного удобства был внедрён заголовок второго уровня:

```
struct TMsgEx: public TMessage {
     int64_t Term;
    int32_t Dst;
   int32_t Src;
};
```

В этот заголовок включены идентификаторы отправителя и получателя в каждом сообщении. За исключением LogEntry, все сообщения наследуются от TMessageEx. LogEntry и AppendEntries реализованы следующим образом:

```
struct log_entry_t : public message_base_t {
       static constexpr msg_type_t message_type = msg_type_t::LOG_ENTRY_TRIE;
    char additional_info[0];
    int32_t term_value;
};
struct TAppendEntriesRequest: public TMsgEx {
    static constexpr EMsgType MsgType
        = EMessageType::APPEND_ENTRIES_REQUEST_TYPE;
    int32_t prev_log_index;
    int32_t prev_log_term;
    int32_t nentries;
};
```

Для упрощения обработки сообщений используется класс под названием MessageHolder, который напоминает shared_ptr:

```
template<typename T>
requires std::derived_from<T, TMsg>
struct msg_holder_t {
    std::shared_ptr<msg_holder_t<T>[]> data_payload;
    std::shared_ptr<char[]> raw_buffer;
    T* message_array[];
    int32_t payload_length;
};
    template<typename U>
    TMsgHolder<U> Cast() {...}

    template<typename U>
    auto maybe() { ... }
};
```


Этот класс включает массив типа char, содержащий само сообщение. Он также может содержать Payload (используемый только для AppendEntry), а также методы для безопасного приведения сообщения базового типа к конкретному (метод Maybe) и небезопасного приведения (метод Cast). Пример использования MessageHolder:

```
void SomeFunction(TMessageHolder<TMsg> msg) {
            auto maybe_append_entries = message.
maybe<TAppendEntriesRequest>();
    if (maybe_append_entries) {
        auto appendEntries = maybe_append_entries.Cast();
    }
    // if we are sure
 auto appendEntries = message.Cast<TAppendEntriesRequest>();
    // usage with overloaded operator->
    auto term = appendEntries->Term;
    auto nentries = appendEntries->Nentries;
    // ...
}
```

И пример из практики в обработчике состояния Candidate:

```
void raft_candidate(Time current_time, MsgHolder<TMsg>
message) {
            if (auto vote_response = message.
maybe<TRequestVoteResponse>()) {
            handle_request_vote(std::move(vote_response.
Cast()));
         } else if (auto vote_request = message.
maybe<TRequestVoteRequest>()) {
        handle_request_vote(current_time, std::move(vote_
request.Cast()));
      } else if (auto append_entries_request = message.
maybe<TAppendEntriesRequest>()) {
     handle_append_entries(current_time, std::move(append_
entries_request.Cast()));
    }
}
```

Этот подход к проектированию повышает эффективность и гибкость обработки сообщений в реализациях алгоритма Raft.

6. Raft-сервер

Теперь рассмотрим реализацию сервера Raft. Сервер Raft будет использовать корутины для сетевых взаимодействий. Сначала рассмотрим корутины, которые обрабатывают чтение и запись сообщений. Примитивы, используемые для этих корутин, обсуждаются далее в работе, наряду с анализом сетевой библиотеки. Корутина записи отвечает за отправку сообщений в сокет, тогда как корутина чтения сложнее. Для чтения она сначала извлекает переменные Type и Len, затем выделяет массив из Len байт и, наконец, читает оставшуюся часть сообщения. Такая структура обеспечивает эффективное и продуктивное управление сетевыми коммуникациями в рамках сервера Raft.

```
template<typename TSocket>
TValueTask<void>
msg_writer_t<TSocket>::write(MsgHolder<TMsg> message) {
         co_await TByteWriter(Socket).Write(message.Mes,
message->Len);
    auto payload_data = std::move(message.Payload);
    for (int32_t index = 0; index < message.payload_size;
index++) {
        co_await write(std::move(payload_data[index]));
    }
    co_return;
}
template<typename TSocket>
TValueTask<MsgHolder<TMsg>> msg_reader_t<TSocket>::read()
{
    decltype(TMsg::Type) message_type;
    decltype(TMsg::Len) message_length;
     auto bytes_read = co_await Socket.ReadSome(&message_
type, sizeof(message_type));
     if (bytes_read != sizeof(message_type)) { /* throw
error */ }
    bytes_read = co_await Socket.ReadSome(&message_length,
sizeof(message_length));
     if (bytes_read != sizeof(message_length)) { /* throw
error */ }
    auto message_holder = NewHoldedMessage<TMsg>(message_
type, message_length);
       co_await TByteReader(Socket).Read(message_holder-
>Value, message_length - sizeof(TMsg));
     auto maybe_append_entries_request = message_holder.
maybe<TAppendEntriesRequest>();
    if (maybe_append_entries_request) {
       auto append_entries = maybe_append_entries_request.
Cast();
       auto number_of_entries = append_entries->Nentries;
       message_holder.InitPayload(number_of_entries);

       for (int32_t i = 0; i < number_of_entries; i++) {
           message_holder.payload[i] = co_await read();
       }
    }
    co_return message_holder;
}
```

Для запуска сервера Raft создаётся экземпляр класса RaftServer и вызывается метод Serve. Метод Serve запускает две корутины. Корутина Idle отвечает за периодическую обработку таймаутов, тогда как InboundServe управляет входящими соединениями.

```
class TRaftServer {
public:
    void Serve() {
        Idle();
        InboundServe();
    }
private:
    TVoidTask InboundServe();
    TVoidTask InboundConnection(TSocket socket);
    TVoidTask Idle();
}
```

Входящие соединения принимаются через вызов accept. После этого запускается корутина InboundConnection, которая считывает входящие сообщения и передает их экземпляру Raft для обработки. Такая конфигурация обеспечивает эффективное управление как внутренними таймаутами, так и внешними коммуникациями в сервере Raft.

```
TVoidTask inbound_serve() {
    while (true) {
        auto client_socket = co_await Socket.Accept();
        inbound_connection(std::move(client_socket));
    }
    co_return;
}
TVoidTask inbound_connection(TSocket client_socket) {
    while (true) {
        auto message = co_await TMsgReader(client_socket).
Read();
          Raft->Process(std::chrono::steady_clock::now(),
std::move(message), client_socket);
                Raft->ProcessTimeout(std::chrono::steady_
clock::now());
        drain_nodes();
    }
    co_return;
}
```

Корутина Idle функционирует следующим образом: она вызывает метод ProcessTimeout каждую секунду ожидания. Следует отметить, что эта корутина использует асинхронное ожидание. Такой подход позволяет серверу Raft эффективно управлять операциями, зависящими от времени, без блокировки других процессов, что улучшает общую отзывчивость и производительность сервера.

```
while (true) {
    Raft->ProcessTimeout(now());
    drain_nodes();
    auto current_time = now();

    if (current_time > t0 + dt) {
        debug_print();
        t0 = t1;
    }
    co_await Poller.Sleep(current_time + sleep_duration);
}
```

Корутина, созданная для отправки исходящих сообщений, отличается своей простотой. Она последовательно отправляет все накопленные сообщения в сокет в цикле. В случае ошибки запускается другая корутина, ответственная за установление соединения (через функцию *connect*). Такая структура обеспечивает плавную и эффективную обработку исходящих сообщений, оставаясь при этом надежной благодаря обработке ошибок и управлению соединениями.

```
try {
    while (!messages.empty()) {
        auto messages_to_send = std::move(messages);
        messages.clear();

        for (auto&& message : messages_to_send) {
                        co_await  TMessageWriter(Socket).
Write(std::move(message));
        }
    }
} catch (const std::exception& exception) {
    reconnect();
}
co_return;
}
```

Приведенные примеры демонстрируют, как корутины значительно упрощают разработку.

Рассмотрим пример запуска кластера Raft с тремя нодами. Каждому экземпляру передается его уникальный идентификатор, а также адреса и идентификаторы других экземпляров. В данном случае клиент взаимодействует исключительно с лидером. Он отправляет случайные строки, при этом поддерживая установленное количество сообщений в процессе передачи и ожидая их фиксации. Такая конфигурация иллюстрирует взаимодействие между клиентом и лидером в многоузловой среде Raft, демонстрируя обработку распределенных данных и консенсуса алгоритмом (рис. 5).

```
Shell
$ ./server --id 1 --node 127.0.0.1:8001:1 --node 127.0.0.1:8002:2 --node 127.0.0.1:8003:3
...
Candidate, Term: 2, Index: 0, CommitIndex: 0,
...
Leader, Term: 3, Index: 1080175, CommitIndex: 1080175, Delay: 2:0 3:0
        MatchIndex: 2:1080175 3:1080175 NextIndex: 2:1080176 3:1080176
....
$ ./server --id 2 --node 127.0.0.1:8001:1 --node 127.0.0.1:8002:2 --node 127.0.0.1:8003:3
...
$ ./server --id 3 --node 127.0.0.1:8001:1 --node 127.0.0.1:8002:2 --node 127.0.0.1:8003:3
...
Follower, Term: 3, Index: 1080175, CommitIndex: 1080175,
...
$ dd if=/dev/urandom | base64 | pv -l | ./client --node 127.0.0.1:8001:1 >log1
 198k 0:00:03 [159.2k/s] [        <=>
```

Р и с. 5. Результаты обработки распределённых данных

F i g. 5. Results of distributed data processing

Затем был проведён замер задержки фиксации *(commit)* для конфигураций кластеров с 3 и 5 узлами. Как и ожидалось, задержка выше для конфигурации с 5 узлами. Результаты представлены в таблице 1.

Т а б л и ц а 1. **Скорость коммитов при разном количестве узлов**

T a b l e 1. **Commit speed with different number of nodes**

Узлы	50-й перцентиль (медиана)	80-й перцентиль	90-й перцентиль	99-й перцентиль
3	293,541 нс	399,667 нс	561,550 нс	38,269 нс
5	422,183 нс	669,539 нс	1,022,557 нс	39,678 нс

Эти результаты подтверждают, что увеличение числа узлов в кластере приводит к увеличению задержки фиксации.

7. Реализация I/O-библиотеки

Теперь рассмотрим библиотеку ввода-вывода, которую была разработана в ходе исследования и теперь используется в реализации сервера Raft. За основу берем стандартную реализацию эхо-сервера:

```
task<> tcp_echo_service() {
    char buffer_data[1024];
    while (true) {
           std::size_t bytes_received = co_await socket.
async_read_some(buffer(buffer_data));
         co_await async_write(socket, buffer(buffer_data,
bytes_received));
    }}
```

Для нашей библиотеки потребовались цикл событий, примитив сокета и методы, такие как read_some/write_some (в нашем случае ReadSome/WriteSome), а также более высокоуровневые обертки *(wrappers)*, такие как async_write/async_read (в нашей реализации TByteReader/TByteWriter).
Для реализации метода ReadSome сокета необходимо создать Awaitable следующим образом:

```
auto read_some(char* buffer, size_t buffer_size) {
    struct awaitable_t {
       bool await_ready() { return false; /* always suspend
*/ }
      void await_suspend(std::coroutine_handle<> handle) {
           poller->AddRead(file_descriptor, handle);
       }
       int await_resume() {
          return read(file_descriptor, buffer_ptr, size);
       }
       TSelect* poller;
       int file_descriptor;
       char* buffer_ptr;
       size_t size;
    };
   return awaitable_t{Poller_, Fd_, buffer, buffer_size};}
```

Когда вызывается co_await, корутина приостанавливается, поскольку await_ready возвращает false. В await_suspend захватывается coroutine_handle и передает его вместе с дескриптором сокета в poller. Когда сокет становится готовым, poller вызывает coroutine_handle для возобновления корутины. При возобновлении вызывается await_resume, который выполняет чтение и возвращает количество прочитанных байт в корутину. Методы WriteSome, Accept и Connect реализованы аналогичным образом.
Poller настраивается следующим образом:

```
struct event_t {
    int file_descriptor;
    int event_type; // READ = 1, WRITE = 2;
    std::coroutine_handle<> coroutine_handle;
};
class select_t {
public:
    void poll_events() {
        for (const auto& event : events) {
            /* FD_SET(read_fds); FD_SET(write_fds); */
        }
      pselect(size, read_fds, write_fds, nullptr, timeout,
nullptr);

        for (int index = 0; index < size; ++index) {
            if (FD_ISSET(index, write_fds)) {
                events[index].coroutine_handle.resume();
            }
            // ...
        }
    }
private: std::vector<event_t> events;};
```

Мы храним массив пар (дескриптор сокета, корутина handle), которые используются для инициализации структур для backend-пуллера (в данном случае, select). Метод Resume вызывается, когда корутины, соответствующие готовым сокетам, пробуждаются.
Это применяется в основной функции следующим образом:

```
simple_task my_task(TSelect& poller) {
    TSocket client_socket(0, poller);
    char data_buffer[1024];

    while (true) {
              auto bytes_read = co_await client_socket.
ReadSome(data_buffer, sizeof(data_buffer));
    }
}
```

Запускается корутина (или несколько корутин), которая переходит в режим ожидания при вызове co_await, после чего управление передается бесконечному циклу, который вызывает механизм поллинга. Если сокет становится готовым в процессе поллинга, соответствующая корутина активируется и выполняется до следующего вызова co_await.
Для чтения и записи сообщений Raft потребовалось создать высокоуровневые обертки над методами ReadSome/WriteSome, аналогичные следующим:

```
template<typename T>
TValueTask<T> read() {
    T output;
    size_t bytes_to_read = sizeof(T);
    char* buffer = reinterpret_cast<char*>(&output);

    while (bytes_to_read > 0) {
        auto bytes_read = co_await Socket.ReadSome(buffer,
bytes_to_read);
        buffer += bytes_read;
        bytes_to_read -= bytes_read;
    }

    co_return output;
}
T item = co_await read<T>();
```

Для реализации этих оберток необходимо создать корутину, которая также функционирует как Awaitable. Корутина состоит из пары: coroutine_handle и promise. coroutine_handle используется для управления корутиной извне, тогда как promise предназначен для внутреннего управления. coroutine_handle может включать методы Awaitable, которые позволяют ожидать результат корутины с помощью co_await.promise используется для хранения результата, возвращаемого с помощью co_return, и для пробуждения вызывающей корутины.
В coroutine_handle, в методе await_suspend, сохраняется coroutine_handle вызывающей корутины. Его значение будет сохранено в promise:

```
template<typename T>
struct TValueTask : std::coroutine_handle<> {
     bool await_ready() { return static_cast<bool>(this-
>promise().value); }

      void await_suspend(std::coroutine_handle<> caller_
handle) {
       this->promise().caller = caller_handle;}
    T await_resume() { return *this->promise().value; }
```

```
    using promise_type = TValuePromise<T>;};
```

В самом promise метод return_value будет хранить возвращаемое значение. Вызывающая корутина пробуждается с помощью awaitable, который возвращается в final_suspend. Это происходит потому, что после вызова co_return компилятор выполняет co_await на final_suspend.

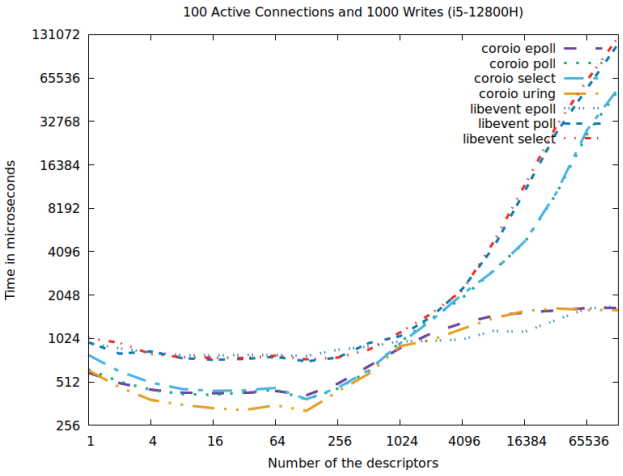
```
template<typename T>
struct TValuePromise {
      void return_value(const T& value) { this->value =
value; }
       std::suspend_never initial_suspend() { return {}; }
    TFinalSuspendContinuation<T> final_suspend() noexcept;
    std::optional<T> value;
   std::coroutine_handle<> caller = std::noop_coroutine();
};
```

В методе await_suspend вызывающая корутина может быть возвращена, и она будет автоматически пробуждена. Важно отметить, что вызванная корутина теперь будет находиться в спящем состоянии, и её coroutine_handle должно быть уничтожено с помощью destroy, чтобы избежать утечки памяти. Это можно осуществить, например, в деструкторе TvalueTask.

```
template<typename T>
struct TFinalSuspendContinuation {
    bool await_ready() noexcept { return false; }
    std::coroutine_handle<> await_suspend(
          std::coroutine_handle<TValuePromise<T>> handle)
noexcept
    {
        return handle.promise().caller;
    }
    void await_resume() noexcept { }
};
```

Результаты исследования

Завершив описание библиотеки, нами был использован тест производительности (бенчмарк) libeven, с помощью которого мы замерили эффективность работы нашей реализации и сформировали график. Этот тест создает цепочку из N Unix pipe, каждый из которых связан с последующим. Затем выполняется 100 операций записи в цепочку, которые продолжаются до достижения 1000 вызовов записи в целом. На приведенном ниже рисунке N показано время выполнения теста в зависимости от N для различных бэкендов библиотеки *(coroio)* по сравнению с libevent. Этот тест демонстрирует, что производительность yfitq библиотеки сопоставима с libevent, подтверждая её эффективность и действенность в управлении операциями ввода-вывода (рис. 6).



Р и с. 6. Результаты тестовых прогонов

F i g. 6. Test Run Results

Обсуждение и заключение

В данной работе описана реализация сервера Raft с использованием корутин C++20. Специально разработанная библиотека ввода-вывода, является ключевым компонентом данной реализации, так как она эффективно управляет асинхронными операциями ввода-вывода. Производительность библиотеки была подтверждена с помощью теста производительности libevent, что продемонстрировало её компетентность.

References

[1] Treiblmaier H., Rejeb A., Strebinger A. Blockchain as a Driver for Smart City Development: Application Fields and a Comprehensive Research Agenda. *Smart Cities.* 2020;3(3):853-872. https://doi.org/10.3390/smartcities3030044

[2] Bahrepour D., Maleki R. Benefit and limitation of using blockchain in smart cities to improve citizen services. *GeoJournal.* 2024;89:57. https://doi.org/10.1007/s10708-024-11040-7

[3] Ongaro D., Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USA: USENIX Association; 2014. p. 305-319. Available at: https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.pdf (accessed 12.05.2024).

[4] Rodrigues L.A., Freitas A.E.S., Duarte Jr. E.P., Fulber-Garcia V. A Hierarchical Adaptive Leader Election Algorithm for Crash-Recovery Distributed Systems. In: Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing (LADC '24). New York, NY, USA: Association for Computing Machinery; 2024. p. 136-145. https://doi.org/10.1145/3697090.3697102

[5] Lu S., Zhang X., Zhao R., Chen L., Li J., Yang G. P-Raft: An Efficient and Robust Consensus Mechanism for Consortium Blockchains. *Electronics.* 2023;12(10):2271. https://doi.org/10.3390/electronics12102271

[6] Albshaier L., Budokhi A., Aljughaiman A. A Review of Security Issues When Integrating IoT With Cloud Computing and Blockchain. *IEEE Access.* 2024;12:109560-109595. https://doi.org/10.1109/ACCESS.2024.3435845

[7] Sati A., Al-Tabtabai H. A paradigm shift toward the application of blockchain in enhancing quality information management. *Construction Innovation: Information Process Management.* 2024;24(1):407-424. https://doi.org/10.1108/CI-05-2023-0099
[8] Dinh T.T.A., Liu R., Zhang M., Chen G., Ooi B.C., Wang J. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Transactions on Knowledge and Data Engineering*. 2018;30(7):1366-1385. https://doi.org/10.1109/TKDE.2017.2781227
[9] Asiamah E.A., Keelson E., Agbemenu A.S., Tchao E.T., Adjaidoo T.-S., Klogo G.S. Optimizing Blockchain Querying: A Comprehensive Review of Techniques, Challenges, and Future Directions. *IEEE Access.* 2024;12:196282-196305. https://doi.org/10.1109/ACCESS.2024.3522584
[10] Bogdanov A., Shchegoleva N., Khvatov V., Kiyamov J., Dik A. Combining PBFT and Raft for Scalable and Fault-Tolerant Distributed Consensus. *Physics of Particles and Nuclei*. 2024;55(3):418-420. https://doi.org/10.1134/S1063779624030225
[11] Zhang P., Zhou M. Security and Trust in Blockchains: Architecture, Key Technologies, and Open Issues. *IEEE Transactions on Computational Social Systems.* 2020;7(3):790-801. https://doi.org/10.1109/TCSS.2020.2990103
[12] Xu R., Chen Y., Blasch E., Chen G. Microchain: A Hybrid Consensus Mechanism for Lightweight Distributed Ledger for IoT. *arXiv:*1909.10948. 2019. https://doi.org/10.48550/arXiv.1909.10948
[13] Kogias M., Bugnion E. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In: Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). New York, NY, USA: Association for Computing Machinery; 2020. Article number: 25. https://doi.org/10.1145/3342195.3387545
[14] Dujak D., Sajter D. Blockchain Applications in Supply Chain. In: Kawa A., Maryniak A. (eds.) SMART Supply Network. *EcoProduction.* Springer, Cham; 2019. p. 21-26. https://doi.org/10.1007/978-3-319-91668-2_2
[15] Reda M., Kanga D.B., Fatima T., Azouazi M. Blockchain in health supply chain management: State of art challenges and opportunities. *Procedia Computer Science*. 2020;175:706-709. https://doi.org/10.1016/j.procs.2020.07.104
[16] Noergaard T. Chapter 6 – Board I/O. In: Embedded Systems Architecture (Second Edition): A Comprehensive Guide for Engineers and Programmers. Amsterdam: Elsevier Science; 2018. p. 261-293. https://doi.org/10.1016/B978-0-12-382196-6.00006-6
[17] Tavares B., Correia F.F., Restivo A. A survey on blockchain technologies and research. *Journal of Information Assurance and Security*. 2019;14:118-128.
[18] Moosavi N., Taherdoost H., Mohamed N., Madanchian M., Farhaoui Y., Khan I.U. Blockchain Technology, Structure, and Applications: A Survey. *Procedia Computer Science*. 2024;237:645-658. https://doi.org/10.1016/j.procs.2024.05.150
[19] Wang R., Zhang L., Xu Q., Zhou H. K-Bucket Based Raft-Like Consensus Algorithm for Permissioned Blockchain. In: 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS). Tianjin, China: IEEE Press; 2019. p. 996-999. https://doi.org/10.1109/ICPADS47876.2019.00152
[20] Gu R., Huang D. A Leadership Transfer Algorithm for the Raft. In: Sun Y., Cai L., Wang W., Song X., Lu Z. (eds.) Blockchain Technology and Application. CBCC 2022. *Communications in Computer and Information Science.* Vol. 1736. Singapore: Springer; 2022. p. 13-30. https://doi.org/10.1007/978-981-19-8877-6_2
[21] Du Z., Qu Z., Fu Y., Huang M., Liu L. Multi-strategy-based leader election mechanism for the Raft algorithm. *Concurrency and Computation: Practice and Experience.* 2023;35(22):e7734. https://doi.org/10.1002/cpe.7734
[22] Wu Y., Wu Y., Liu Y., Shi T. The research of the optimized solutions to Raft consensus algorithm based on a weighted PageRank algorithm. In: 2022 Asia Conference on Algorithms, Computing and Machine Learning (CACML). Hangzhou, China: IEEE Press; 2022. p. 784-789. https://doi.org/10.1109/CACML55074.2022.00135
[23] Zhang P., Tao Y., Zhao Q., Zhou M. A Rate-and-Trust-Based Node Selection Model for Block Transmission in Blockchain Networks. *IEEE Internet of Things Journal*. 2023;10(2):1605-1616. https://doi.org/10.1109/JIOT.2022.3210197
[24] Zhang P., Schmidt D.C., White J., Dubey A. Chapter Seven-Consensus mechanisms and information security technologies. *Advances in Computers.* 2019;115:181-209. https://doi.org/10.1016/bs.adcom.2019.05.001
[25] Auhl Z., Chilamkurti N., Alhadad R., Heyne W. A Comparative Study of Consensus Mechanisms in Blockchain for IoT Networks. *Electronics.* 2022;11(17):2694. https://doi.org/10.3390/electronics11172694

Поступила 12.05.2024; одобрена после рецензирования 18.07.2024; принята к публикации 02.09.2024.
Submitted 12.05.2024; approved after reviewing 18.07.2024; accepted for publication 02.09.2024.

Об авторах:
Мельников Максим Олегович, аспирант кафедры математического моделирования, компьютерных технологий и информационной безопасности Института математики, естествознания и техники, ФГБОУ ВО «Елецкий государственный университет им. И. А. Бунина» (399770, Российская Федерация, Липецкая область, г. Елец, ул. Коммунаров, д. 28-1), **ORCID: https://orcid.org/0000-0003-1921-3033,** melnikov.maxx@yandex.ru
Игонина Елена Викторовна, заведующий кафедрой математики и методики её преподавания Института математики, естествознания и техники, ФГБОУ ВО «Елецкий государственный университет им. И. А. Бунина» (399770, Российская Федерация, Липецкая область, г. Елец, ул. Коммунаров, д. 28-1), кандидат физико-математических наук, доцент, **ORCID: https://orcid.org/0000-0002-7369-6219,** elenaigonina7@mail.ru

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Maxim O. Melnikov, Postgraduate student of the Chair of Mathematical Modeling, Computer Technologies and Information Security, Institute of Mathematics, Natural Science and Technology, Bunin Yelets State University (28-1 Kommunarov St., Yelets 399770, Lipetsk region, Russian Federation), **ORCID: https://orcid.org/0000-0003-1921-3033,** melnikov.maxx@yandex.ru

Elena V. Igonina, Associate Professor, Head of the Chair of Mathematics and Methods of its Teaching, Institute of Mathematics, Natural Science and Technology, Bunin Yelets State University (28-1 Kommunarov St., Yelets 399770, Lipetsk region, Russian Federation), Cand. Sci. (Phys.-Math.), Associate Professor, **ORCID: https://orcid.org/0000-0002-7369-6219,** elenaigonina7@mail.ru

All authors have read and approved the final manuscript.