**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

**Факультет Информатика и системы управления» (ИУ)**

**Кафедра «Информационная безопасность» (ИУ8)**

# Отчет

# по научно-исследовательской работе студента

на тему Анализ и формальная верификация алгоритма консенсуса Raft

ФИО студента: Железцов Никита Владимирович

Группа: ИУ8-94-2024 Л.д.: 20У474

Специальность: 10.05.01 Компьютерная безопасность

Специализация: 10.05.01_01 Математические методы защиты информации

Научный руководитель НИРС доцент кафедры ИУ8 Колесников Александр Владимирович

Работа выполнена _____ Н.В.Железцов
дата подпись студента И.О.Фамилия студента

Допуск к защите _____ А.В.Колесников
дата подпись научного руководителя И.О.Фамилия научного руководителя

Отчет принят _____ Д.О.Левиев
дата подпись ответственного за НИРС И.О.Фамилия Ответственного за НИРС

| Результаты защиты НИРС | | | |
|---|---|---|---|
| Дата | Балл | Подпись | ФИО |
| | | | |
| | | | |
| | | | |

Москва
2024

## РЕФЕРАТ

Отчёт содержит 50 стр., 23 рис., 13 источн., 2 прил.

Ключевые слова: распределенные системы, задача двух генералов, невозможность Фишера-Линча-Патерсона, консенсус, Raft, Paxos, язык спецификаций TLA+.

Основная цель работы — провести анализ алгоритма консенсуса Raft, описать принципы его работы, сопоставить его с другими невизантийскими алгоритмами и проверить его корректность с использованием языка спецификаций TLA+.

В процессе работы было проведено исследование отечественной и зарубежной литературы по заданной теме, изучены подходы к достижению консенсуса в распределенных системах. Была написана TLA+ спецификация одной из имплементаций алгоритма Raft, в частности Raft с реконфигурацией реплик.

В результате данного исследования было установлено, что Raft проще для понимания и реализации, но принципиального отличия между Raft и Paxos нет, а потому выбор за разработчиками. Было установлено, что дизайн Raft соответствует предъявляемым требованиям.

## СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Мир вычислительных технологий за свою жизнь пережил значительные изменения: от монолитных приложений прошлых лет до современных микросервисов - подходы к обработке данных претерпели глубокую трансформацию. Централизованные приложения, некогда считавшиеся вершиной технологий, уже не отвечали запросам времени, и цифровая реальность потребовала чего-то более гибкого, масштабируемого и устойчивого.

Так наступила эпоха распределенных систем. Они разделяют задачи на части, распределяя их между множеством узлов, работающих в гармонии друг с другом. Любое высоконагруженное приложение или база данных использует распределенные системы для обработки миллионов одновременных запросов.

Однако под всей этой кажущейся простотой скрывается фундаментальная проблема: необходимость согласования действий и состояний множества узлов, разбросанных по разным частям мира и часто подверженных сбоям. Здесь на помощь и приходят алгоритмы консенсуса, которые в распределенных системах выступают хранителями целостности данных, гарантами отказоустойчивости.

### Обоснование актуальности темы исследования

В современных распределённых системах алгоритмы консенсуса (такие как Raft, Paxos) являются основой для обеспечения согласованности данных и состояний между узлами. Учитывая растущую зависимость от распределённых систем в различных отраслях, корректность этих алгоритмов критически важна для обеспечения надёжности и доступности сервисов.

Формальная верификация предоставляет математически строгие методы проверки алгоритмов. Использование TLA+ для анализа и проверки алгоритма Raft позволяет доказать корректность его ключевых свойств. Это снижает вероятность ошибок и позволяет проверить корректность дизайна алгоритма. Только после уверенности в алгоритме можно приступать к его имплементации в коде.

Верификация алгоритмов распределённых систем становится всё более востребованной в сфере информационных технологий. Компании, такие как Amazon, Google, и Microsoft, активно используют формальные методы для проверки своих систем. Исследование формальной верификации Raft с применением TLA+ предоставляет актуальные знания и навыки, востребованные на рынке труда.

Таким образом, анализ и формальная верификация алгоритма Raft с использованием TLA+ представляют собой актуальную тему, находящуюся на пересечении интересов академического сообщества и индустрии IT. Результаты исследования будут полезны как исследователям, изучающим механизмы согласования в распределённых системах, так и практикующим разработчикам, создающим надёжные и отказоустойчивые системы.

### Цели и задачи НИРС

Целью исследования является изучение алгоритма Raft, его формальный анализ и верификация. Для достижения этой цели предполагается решение следующих задач:

- изучение и описание принципов работы Raft;
- сравнительный анализ Raft и других алгоритмов консенсуса;
- формальная спецификация алгоритма Raft на TLA+ и проверка его корректности.

В рамках данной работы рассматриваются только невизантийские алгоритмы консенсуса.

# ОСНОВНАЯ ЧАСТЬ

В первой части задается основа для последующего анализа алгоритмов консенсуса, определяются термины распределенной системы и консенсуса, задаются условия и ограничения, в которых эти алгоритмы могут быть применены.

Во второй части приводится описание алгоритма консенсуса Paxos, в третьей - Raft. Оба этих алгоритма выполняются в распределенной системе, описанной в первой части.

В четвертой же части дается определение TLA+, проводится формальная верификация алгоритма консенсуса Raft, строится модель система и производится ее проверка.

# 1 Модель распределенной системы

Формального определения распределенной вычислительной системы в настоящее время не существует. Из множества различных определений, можно выделить ироничное определение Лесли Лампорта, которое он дал в мае 1987 года, в своем письме коллегам по поводу очередного отключения электроэнергии в машинном зале:

«Распределенной вычислительной системой можно назвать такую систему, в которой отказ компьютера, о существовании которого вы даже не подозревали, может сделать ваш собственный компьютер непригодным к использованию».

Эндрю Таненбаум предложил следующее, более серьзное, определение [1]:

«Распределенная вычислительная система (РВС) – это набор соединенных каналами связи независимых компьютеров, которые с точки зрения пользователя некоторого программного обеспечения выглядят единым целым».

Именно это определение и будет использоваться в данной работе. Таким образом, в РВС есть несколько автономных участников (иногда называемых процессами, узлами или репликами). Каждый участник обладает своим локальным состоянием. Участники выполняют некий алгоритм, общаются, обмениваясь сообщениями через каналы связи между ними. Вне системы существуют пользователи, которые могут отправлять запросы к различным узлам системы и ожидать от них ответов.

## 1.1 Модель каналов связи

Связь через каналы часто ненадежна: сообщения могут теряться, задерживаться или приходить в неправильном порядке. В качестве отправной точки взят канал с приемлемыми потерями (fair-loss) [2]:

- Допустимые потери. Если отправитель и получатель функционируют корректно и процесс отправляет сообщение бесконечно много раз, оно в конечном итоге будет доставлено бесконечное число раз.
- Ограниченное дублирование. Одно отправленное сообщение не будет доставлено бесконечное количество раз.
- Без создания. Канал не создает новых сообщений, то есть доставляются только те сообщения, которые были отправлены.

Канал с приемлемыми потерями является полезной абстракцией и первым строительным блоком для протоколов обмена данными с сильными гарантиями. По своей сути он похож на протокол UDP, который позволяет отправлять сообщения от одного процесса другому, но не предлагает надежной семантики доставки на уровне протокола. Однако гарантии, предоставляемые такой моделью недостаточны для построения надежных распределенных систем.

Для повышения надежности связи можно использовать подтверждения (acknowledgement, ACK), позволяющие получателю уведомить отправителя о доставке сообщения. Для этого применяются полнодуплексные каналы связи и добавляются механизмы, которые помогают различать сообщения, например, уникальные порядковые номера (sequence numbers), монотонно возрастающие идентификаторы.

До тех пор, пока отправитель не получит подтверждение о доставке сообщения, он не может быть уверен, было ли оно обработано, будет ли обработано в будущем, утеряно, либо удаленный процесс вышел из строя до его получения. Отправитель может повторно передать сообщение, но это может вызвать его дублирование. Безопасная обработка дубликатов возможна только в случае, если выполняемая операция является идемпотентной. Поскольку в реальных условиях не всегда удается обеспечить идемпотентность операций, требуется применять эквивалентные гарантии. Для этого можно использовать методы дедупликации (deduplication), предотвращающие повторную обработку одного и того же сообщения.

Повторные передачи и несоблюдение порядка доставки могут приводить к неверному порядку поступления сообщений. Благодаря введению порядковых номеров получатель может использовать их для восстановления порядка и обеспечения обработки сообщений по принципу FIFO (first-in, first-out).

Применив все описанные выше ограничения к каналу с приемлемыми ограничениями, получаем совершенный канал (perfect link), предоставляющий гарантии [2]:

- Надежная доставка. Каждое сообщение, отправленное один раз корректным процессом А корректному процессу Б, в конце концов будет доставлено.
- Порядок сообщений. Сообщения будут доставлены в том порядке, в котором они были отправлены.

– Без дублирования. Ни одно сообщение не будет обработано более одного раза.
– Без создания. Канал доставляет только отправленные сообщения, не создавая новые.

Именно совершенный канал будет использоваться как модель канала для построения алгоритмов коненсуса распределенных систем. Он похож на протокол TCP.

Однако даже совершенный канал не защищает алгоритм от ситуации разделения сети, под которой понимается, что два или более процесса не могут связаться друг с другом. Независимые группы процессов могут продолжать выполнение алгоритмов и выдывать противоречивые результаты, могут происходить асимметричные отказы каналов, при которых сообщения могут проходить только в одну сторону, но не обратно. Все это необхоимо учитывать при разработке алгоритмов в распределенных системах.

## 1.2 Определение консенсуса

Алгоритм консенсуса описывает работу распределенной системы, которая при наличии нескольких процессов, начинающих работу с некого начального состояния, переводит все процессы в одинаковое состояние. Чтобы алгоритм консенсуса был корректным, должны выполняться следующие условия [3]:

– Согласованность. Принимаемое протоколом решение должно быть единодушным: каждый процесс выбирает некоторое значение, которое должно быть одинаковым для всех процессов.
– Действительность. Согласованное значение должно быть предложено одним из процессов. Т.е. это не может быть некое произвольное значение.
– Окончательность. Согласованность принимает окончательный характер после того, как уже не остается процессов, не достигших состояния принятия решения.

## 1.3 Теорема Фишера-Линча-Патерсона

Мысленный эксперимент, известный как «задача двух генералов», наглядно иллюстрирует сложности обеспечения консенсуса в распределенных системах. Эксперимент демонстрирует, что в асинхронной системе, где нет временных ограничений на доставку и ответы, невозможно достичь полного согласия между сторонами, даже с использованием многочисленных подтверждений.

Суть задачи такова: две армии под командованием генералов расположены по разные стороны от города и должны атаковать одновременно, чтобы достичь успеха. Генералы передают сообщения через посланников, чтобы договориться об атаке. Однако посланники могут быть перехвачены или не доставить сообщение, что создает неопределенность. То есть генералам необходимо достичь консенсуса по времени начала атаки.

Например, генерал А отправляет сообщение MSG(N), предлагая атаку в определенное время. Генерал Б, получив сообщение, посылает подтверждение ACK(MSG(N)). На рис. 1 показано, что сообщение отправляется в одну сторону и подтверждается другой стороной.

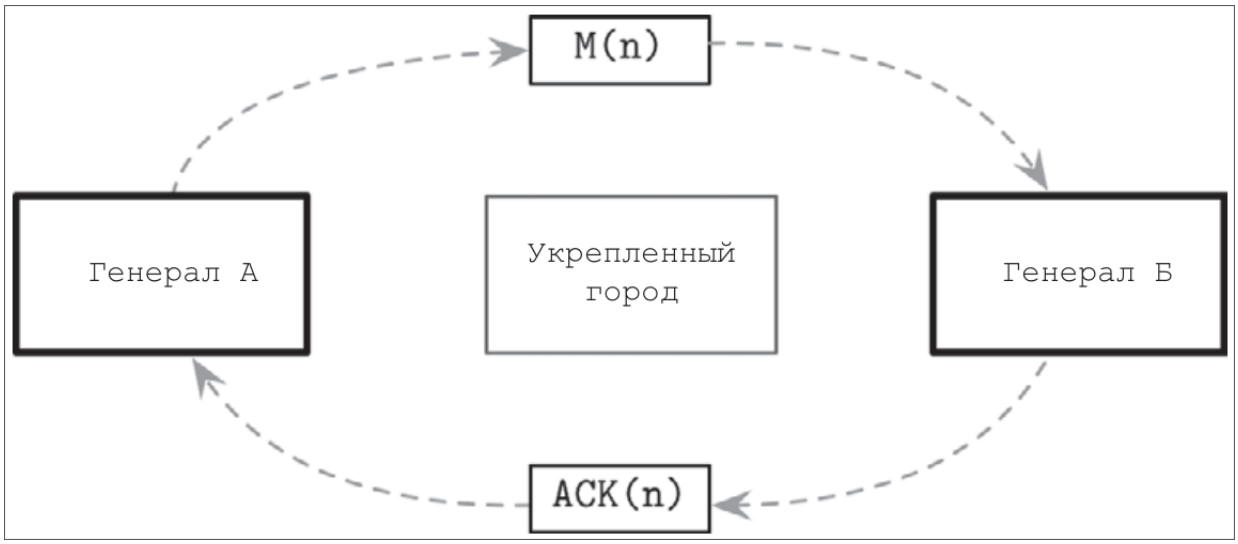


Рисунок 1 – Задача двух генералов

Однако генерал А не может быть уверен, что это подтверждение дошло, оно может быть утеряно. Чтобы устранить сомнения, требуется второе подтверждение — ACK(ACK(MSG(N))). Этот процесс может продолжаться бесконечно, поскольку всегда остается риск того, что последнее сообщение не достигло адресата.

В задаче не сделано предположений о времени, не установлено лимита, за которое генералы должны ответить, связь полностью асинхронна.

В работе Фишера, Линча и Патерсона рассматривается проблема, известная как невозможность ФЛП (FLP Impossibility) [4]. Эта теорема, также именуемая теоремой Фишера–Линча–Патерсона, изучает консенсус в системах, где процессы начинают с начального значения и стремятся согласовать новое общее значение. После выполнения алгоритма это значение должно быть одинаковым для всех корректно работающих процессов.

В исследовании предполагается полностью асинхронная среда, где процессы не имеют общего времени. Алгоритмы в таких системах не могут полагаться на время ожидания, а у процесса отсутствует возможность определить, отказал ли другой процесс или просто работает медленно. В статье доказывается, что при указанных предположениях невозможно создать протокол, который гарантирует достижение консенсуса за ограниченное время. Даже один сбой процесса, не сопровождающийся уведомлением, делает консенсус недостижимым для полностью асинхронной распределенной системы.

Тем не менее, теорема ФЛП не утверждает, что достижение консенсуса в принципе невозможно. Она лишь показывает, что в асинхронной системе консенсус не всегда достижим за конечное время. В реальных системах часто присутствует некоторая степень синхронности, и эту особенность нужно учитывать.

### 1.4 Синхронность

Из теоремы ФЛП следует, что ключевой характеристикой распределенной системы является предположение о времени. В асинхронных системах невозможно гарантировать ограниченные задержки в доставке сообщений, упорядоченность их получения. Процесс выдает ответ через неопределенной долгое время.

Однако, одним из аргументов против асинхронных систем является их оторванность от реальности: процессы не имеют произвольно разные скорости обработки, а задержки передачи сообщений не бывают бесконечными.

Можно сделать предположения менее строгими, считая систему синхронной. Может предполагаться, что процессы работают с сопоставимыми скоростями, задержки передачи сообщений в канале ограничены, их доставка не может занимать бесконечно долгое время.

В модель синхронной системы также можно добавить локальные для процесса синхронизированные часы: при этом существует некоторая верхняя граница в раз нице во времени между двумя локальными для процессов источниками времени [2].

Свойства асинхронных и синхронных моделей можно объединить, рассматривая систему как частично синхронную. Частично синхронная система обладает некото рыми свойствами синхронной системы, но при этом ограничения на время доставки сообщений, уход показаний часов и относительные скорости обработки могут быть приблизительными и действовать лишь в большинстве случаев.

## 1.5 Модели отказов

Модель отказов описывает, каким образом могут происходить сбои в работе процессов распределенной системы. Например, можно предположить, что процесс может полностью завершить работу и не восстановиться, восстановиться через некоторое время или начать выдавать некорректные данные из-за сбоя или преднамеренно.

Поскольку процессы в распределенных системах взаимодействуют друг с другом при выполнении алгоритма, сбои в одном из них могут нарушить выполнение всей системы. Для разработки алгоритмов в распределенных системах необходимо понимать, какие отказы могут случиться, чтобы правильно обрабатывать каждый из них.

### Аварийное завершение

В этой модели предполагается, что процесс перестает выполнять шаги алгоритма и не отправляет сообщений другим процессам. После сбоя процесс остается в этом состоянии и больше не участвует в текущем выполнении алгоритма.

Важно отметить, что эта модель не запрещает восстановление процессов. Процесс может восстановиться, синхронизироваться с текущим состоянием системы и участвовать в новых раундах алгоритма.

Процесс, который восстанавливается после сбоя, может продолжить выполнение с последнего известного ему шага. Алгоритмы, поддерживающие восстановление, должны вводить в систему понятия устойчивого состояния алгоритма и протокола восстановления [5].

### Пропуск

В этой модели процесс пропускает выполнение отдельных шагов алгоритма, либо эти шаги невидимы для других участников, либо процесс не может отправлять и получать сообщения. Пропуски включают сетевые сбои, вызванные неисправностями каналов связи, отказами оборудования или перегрузкой сети.

Пропуски возникают, когда алгоритм не завершает определенные действия, или их результаты не достигают других процессов. Например, потерянное сообщение может привести к тому, что отправитель считает его доставленным, несмотря на его окончательную утрату.

### Византийские ошибки

Произвольные, или византийские, ошибки (Byzantine faults) представляют собой наиболее сложный класс отказов. В этом случае процесс продолжает выполнять шаги алгоритма, но делает это некорректно.

Такие сбои могут быть вызваны ошибками в программном обеспечении, выполнением разных версий алгоритма или намеренными действиями злоумышленников. Например, в распределенных системах без централизованного управления, таких как криптовалюты, процессы могут фальсифицировать данные, чтобы ввести систему в заблуждение.

### Обработка отказов

Одним из способом обработки отказов при невизантийских ошибках является введение в алгоритм избыточности. Таким образом отказ может быть замаскирован: даже если один или несколько процессов откажут, то пользователь этого не заметит [6]. Raft и Paxos, которые будут рассмотрены в данной работе, основываются на модели отказов и минимизирует влияние отказов путем использования избыточности процессов.

В случае работы в распределенной системе с византийскими ошибками используется перекрестная проверка других узлов на каждом шаге, поскольку узлы не могут полагаться друг на друга или на лидера и должны проверять поведение других узлов, сравнивая возвращаемые результаты с ответами большинства. PBFT, PoW, PoS борются с ошибками именно так.

## 1.6 Итоги

Таким образом, к распределенной системе, в которой будет исполняться любой из описанных в данной работе алгоритм консенсуса, применяются следующие ограничения:

- Система передает сообщения по совершенным каналам, однако допустимо разделение сети.
- Необходимо учитывать возможность отказа узлов: аварийное завершение и пропуск части алгоритма процессом.
- В системе невозможны византийские ошибки. Предполагается, что все узлы честные и выполняют свои функции корректно.
- Система является частично синхронной, в ней существует понятие времени, которое используется для синхронизации локальных часов. Сообщения в системе могут отправляться бесконечно, процессы могут работать с произвольной скоростью.

## 2 Паксос

Алгоритм Паксос (Paxos) считается одним из самых известных методов достижения консенсуса. Его предложил Лесли Лэмпорт в своей работе The Part-Time Parliament («Парламент с неполной занятостью») [7]. В данной статье идея консенсуса была представлена через метафору законодательного процесса и голосования, происходящих на острове Паксос. Позднее, в 2001 году, Лэмпорт опубликовал статью Paxos Made Simple («Простой Паксос») [8], где использовал упрощенную терминологию, которая и используется в данной работе.

В рамках алгоритма Паксос участники могут выполнять одну из трех ролей:

- Заявители (proposers). Принимают значения от клиентов, формируют предложения для их утверждения и пытаются получить голоса акцепторов.
- Акцепторы (acceptors). Участвуют в голосовании за принятие или отклонение предложений, сформированных заявителями. Для устойчивости к отказам система включает несколько акцепторов, однако для принятия решения достаточно кворума (то есть большинства голосов).
- Ученики (learners). Отвечают за хранение результатов принятых решений, выполняя роль реплик.

  В большинстве реализаций эти роли совмещены в одном процессе.

Каждое предложение содержит уникальный номер, который монотонно увеличивается, и значение, предложенное клиентом. Номер предложения используется для установления общего порядка операций и определения их последовательности («до» или «после»). Номера предложений часто оформляются как пара (идентификатор узла и временная метка). Идентификаторы узлов являются упорядочиваемыми и могут быть использованы для разрешения конфликтов между временными метками. Для корректной работы алгоритма требуется синхронизация локальных часов.

### 2.1 Алгоритм Paxos

Алгоритм Паксос можно разделить на два ключевых этапа: голосование (или этап предложения) и репликацию. На этапе голосования заявители борются за установление своего лидерства, а на этапе репликации заявитель распространяет согласованное значение среди акцепторов.

Заявитель выступает как начальная точка контакта для клиента, получая значение, которое должно быть принято, и пытается собрать голоса от акцепторов. После этого акцепторы рассылают информацию о принятом значении среди учеников, что позволяет ответить пользователю. Ученики, в свою очередь, увеличивают коэффициент репликации согласованного значения.

Только один заявитель может получить большинство голосов. В случае, если голоса распределяются равномерно, ни один из заявителей не сможет набрать большинство, и тогда им придется начать процесс заново.

На этапе предложения заявитель отправляет запрос $(n)$ (где $n$ — номер предложения) большинству акцепторов, пытаясь собрать их голоса. Акцептор, получив запрос, должен ответить, соблюдая несколько инвариантов [8]:

- Если акцептор еще не отвечал на запрос с более высоким порядковым номером, он обещает не принимать предложения с меньшими номерами.
- Если акцептор уже принял какое-то предложение, он уведомляет заявителя о принятом значении через сообщение $(m, v_{\text{принятых}})$.
- Если акцептор ранее ответил на запрос с большим порядковым номером, он уведомляет заявителя о существующем предложении с более высоким номером.
- Акцептор может ответить на несколько запросов, если последний имеет наибольший порядковый номер.

Когда заявитель получает большинство голосов, начинается этап репликации. Заявитель фиксирует предложение, отправляя акцепторам сообщение $(n, v)$, где $v$ — это значение, соответствующее предложению с наибольшим номером среди полученных ответов от акцепторов, или собственное значение заявителя, если акцепторы не приняли других предложений.

Акцептор примет предложение с номером $n$, если на этапе предложения он не ответил на запрос с большим номером. Если акцептор отклоняет предложение, он отправляет заявителю максимальный порядковый номер, который он видел, для актуализации данных заявителя [8].

Обобщенная схема раунда Паксос приведена на рис 2.

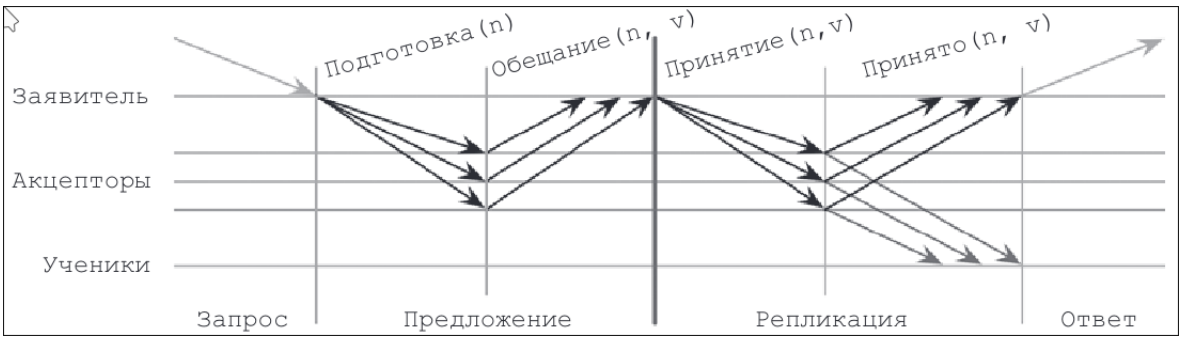


Рисунок 2 – Схема раунда Паксос

После того как консенсус относительно значения достигнут (хотя бы один акцептор принял решение), последующие заявители должны принять это значение, чтобы сохранить согласованность. Поэтому акцепторы возвращают последнее принятое значение. Если ни один акцептор не видел предыдущего значения, заявителю разрешается выбрать собственное.

Ученики получают информацию о принятом значении, когда большинство акцепторов сообщает им об этом. Акцепторы могут сразу уведомить учеников о принятом значении. При наличии нескольких учеников акцепторы должны уведомить каждого, однако можно выделить несколько учеников, которые будут отвечать за уведомление других.

Таким образом, основной целью первого этапа алгоритма является установление лидера и определение принятого значения, что позволяет перейти ко второму этапу — распространению значения. В базовом алгоритме оба этапа выполняются каждый раз, когда необходимо принять решение. Однако на практике можно уменьшить количество шагов. Этот подход будет рассмотрен позже в разделе "Мульти-Паксос".

### 2.2 Мульти-Паксос

Ранее мы рассматривали классический Паксос, в котором любой узел может стать заявителем и инициировать раунд. Проблема этого подхода в том, что для каждого нового раунда репликации требуется запускать стадию предложения (этап с Подготовкой). Чтобы избежать таких повторений и дать заявителю возможность многократно использовать своё положение, применяется мульти-Паксос [8]. Он вводит роль лидера — выделенного заявителя, который после утверждения может сразу приступать к репликации, минуя стадию предложения.

В классическом Паксосе чтение часто реализуется через дополнительный раунд, собирающий любые незавершённые данные, поскольку нет гарантии, что последний заявитель имеет самую свежую версию состояния. В мульти-Паксосе появляется сходная проблема: если мы читаем данные у лидера, который уже успел смениться, то можем получить устаревшую информацию. Чтобы этого избежать и поддерживать линеаризуемость, некоторые реализации используют механизм аренды (lease) [9]. Лидер периодически подтверждает свою активность, а узлы обещают не принимать другие предложения в течение срока аренды. Этот приём не гарантирует абсолютную корректность, но ускоряет операции чтения при условии ограниченной синхронизации часов. Однако в случае рассинхронизации часов, когда лидер считает, что его срок аренды не истек, а другие участники - наоборот, линеаризуемость недостижима.

Мульти-Паксос обычно описывают как реплицируемый журнал операций над некоторой структурой. При сбоях участники восстанавливают данные из долговременного журнала сообщений. Чтобы журнал не разрастался бесконечно, принято периодически создавать моментальные снимки состояния (снапшоты) и усекать журнал до отметки этого снимка.

## 3 Raft

На протяжении длительного времени для достижения консенсуса применялся алгоритм Паксос, однако в кругах разработчиков распределённых систем он считался чрезмерно сложным. В 2013 году был предложен новый подход под названием Raft, ориентированный на упрощение понимания и реализации [10].

В Raft каждый узел хранит локально журнал команд, исполняемых конечным автоматом. Так как все процессы получают одинаковые входные данные и применяют идентичные команды в одном и том же порядке, их конечные автоматы приходят к одинаковому состоянию. Одно из отличий Raft заключается в том, что роль лидера здесь вынесена на первый план: он координирует репликацию и манипуляции над конечным автоматом. С этой точки зрения Raft схож с Мульти-Паксосом и атомарной рассылкой: среди узлов выбирается лидер, который принимает решения и задаёт упорядочение сообщений.

Алгоритм Raft определяет три основные роли:

- Кандидат (candidate): Узел, который пытается стать лидером. Он набирает голоса большинства узлов. Если выборы не приводят к явному победителю, запускается новый период и процесс голосования повторяется.
- Лидер (leader): Временный управляющий кластером, обрабатывающий запросы клиентов и взаимодействующий с реплицируемым конечным автоматом. Лидер выбирается на определённый период, который идентифицируется возрастающим номером. Если лидер перестаёт отвечать или подозревается в отказе, начинается процедура переизбрания.
- Последователь (follower): Пассивный участник, хранящий записи журнала и реагирующий на запросы от лидера и кандидатов. По сути, в Raft он объединяет в себе функции акцептора и ученика из Паксоса. Каждый узел стартует в роли последователя.

Чтобы добиться упорядочения без жёсткой синхронизации часов, в Raft используются периоды (эпохи, термы), в течение которых существует только один лидер. Каждый период имеет уникальный номер, а команды внутри периода получают дополнительный индекс. Узлы могут по-разному воспринимать текущий период (например, если они пропустили этап выборов), но каждая отправляемая команда указывает номер периода [10]. Если узел видит период с более высоким номером, он обновляет своё значение периода.

Процесс выбора лидера инициируется, когда последователь не получает подтверждений от текущего лидера, полагая, что тот вышел из строя. В этом случае последователь переходит в состояние кандидата и собирает голоса большинства узлов, стремясь стать новым лидером.

На рис. 3 приведена схема раунда Raft.

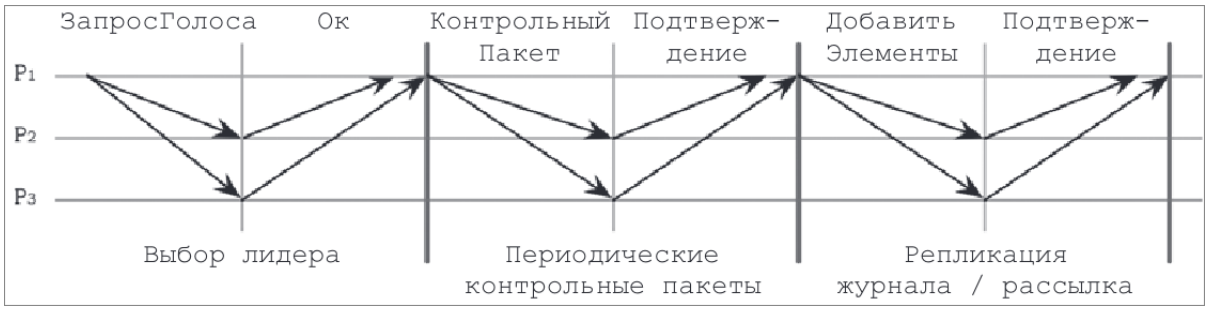


Рисунок 3 – Схема раунда Raft

- Выбор лидера. Когда узел-кандидат (P1 на рисунке) решает стать лидером, он рассылает остальным участникам сообщение , содержащее свой период, последнюю известную ему информацию о периоде, а также идентификатор самой свежей записи в журнале, которую он видел. Если кандидат получает большинство голосов, он становится лидером на текущий период. При этом каждый узел может отдать голос лишь одному кандидату.
- Периодические контрольные пакеты. Для поддержания жизнеспособности системы лидер с определённой периодичностью отправляет контрольные пакеты всем последователям, тем самым подтверждая своё лидерство. Если последователь не получает такие пакеты в течение «тайм-аута выборов», он предполагает сбой лидера и инициирует новый процесс голосования.
- Репликация. Лидер может неоднократно пополнять реплицируемый журнал, отправляя сообщение , где указывает период лидера, индекс и период последней зафиксированной записи, а также одну или несколько новых записей для сохранения.

### 3.1 Роль лидера в Raft

Лидер может быть выбран только среди узлов, содержащих все актуальные записи. Если в процессе выборов журнал последователя более актуальный, чем у кандидата, то голос за этого кандидата не отдается.

Для победы в голосовании кандидат должен получить большинство голосов. Поскольку записи реплицируются строго по порядку, достаточно сравнить идентификаторы последних записей. После избрания лидер начинает принимать запросы от клиентов и реплицирует их на своих последователей. Для этого он добавляет запись в свой журнал и одновременно отправляет её всем последователям.

Когда последователь получает сообщение о добавлении записей, он вносит эти записи в локальный журнал и отправляет подтверждение, сообщая лидеру, что данные сохранены. Как только лидер получает достаточное количество подтверждений, запись считается зафиксированной и помечается соответствующим образом в его журнале.

Поскольку лидером может стать только узел с наиболее актуальными данными, последователь не отправляет ему обновления. Записи журнала передаются только в одном направлении — от лидера к последователям.

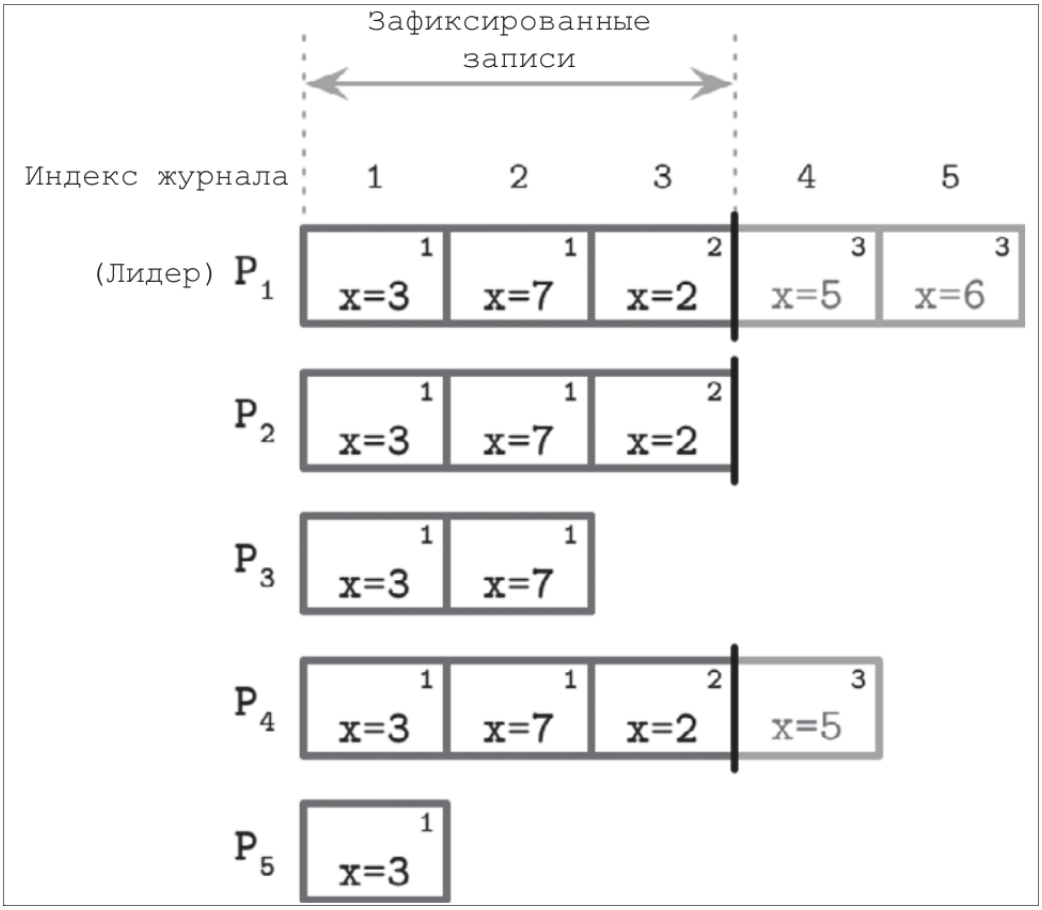


Рисунок 4 – Конечный автомат алгоритма Raft

На рисунке 4 представлен пример раунда достижения консенсуса, в котором узел P1 выступает в роли лидера с наиболее актуальной информацией. Лидер выполняет алгоритм, реплицируя записи на своих последователей и фиксируя их после получения подтверждений. Фиксация одной записи автоматически фиксирует все предшествующие записи в журнале. Решение о фиксации может принимать только лидер. Каждая запись в журнале имеет идентификатор периода (терма, указан в верхнем правом углу записи) и индекс, определяющий её позицию в журнале. Зафиксированные записи гарантированно реплицируются на кворум узлов, что позволяет безопасно применять их к конечному автомату в порядке их добавления.

### 3.2 Сценарии отказов

Когда несколько последователей решают стать кандидатами, но ни один из них не может набрать большинство голосов, такая ситуация называется "разделенным голосованием". Чтобы снизить вероятность таких случаев, алгоритм Raft применяет рандомизированные таймеры. Это позволяет одному из кандидатов начать следующий этап выборов раньше других, получить достаточное количество голосов и быть избранным, пока остальные кандидаты находятся в ожидании. Такой подход ускоряет процесс выборов, исключая необходимость дополнительной координации между кандидатами.

Если последователи отключаются или задерживают ответы, лидер обязан предпринимать дополнительные попытки доставки сообщений. Если подтверждение от узлов не поступает в ожидаемый срок, лидер повторно отправляет сообщения.

Благодаря уникальным идентификаторам, присваиваемым реплицируемым записям, порядок в журнале остается неизменным, даже при повторной доставке сообщений. Последователи устраняют дублирующие записи, основываясь на их порядковых номерах, что предотвращает нежелательные эффекты от повторных отправок. Также порядковые номера используются для соблюдения хронологии в журнале: последователь отклоняет записи с более высокими номерами, если предыдущие записи не совпадают с его журналом. Если две записи из разных журналов имеют одинаковые идентификаторы и индексы, то они хранят одну и ту же команду, а все предшествующие им записи идентичны.

Для обнаружения сбоев лидер отправляет последовательным узлам контрольные сообщения, подтверждая тем самым активность своего периода. Если один из узлов замечает, что текущий лидер перестал отвечать, он инициирует процедуру выборов. Новый лидер восстанавливает состояние кластера, определяя последнюю согласованную запись (то есть запись с наибольшим номером, которую разделяют лидер и последователь). Он приказывает узлам удалить все незафиксированные записи после этой точки и реплицирует актуальные записи из своего журнала. Лидер не удаляет и не перезаписывает собственные записи, а только добавляет новые.

Таким образом, Raft предоставляет следующие гарантии:

- Только один лидер может быть избран одновременно на заданный период (терм); в течение одного периода не может быть двух активных лидеров;
- Лидер не удаляет и не переупорядочивает содержимое журнала; он только добавляет новые сообщения к нему;
- Зафиксированные записи в журнале гарантированно присутствуют в журналах для последующих лидеров;
- Все сообщения однозначно идентифицируются по идентификаторам сообщений и периодов; ни текущий, ни последующие лидеры не могут повторно использовать один и тот же идентификатор для другой записи.

## 4 Реализация алгоритма Raft на TLA+

TLA+ — язык спецификаций, основанный на теории множеств, логике первого порядка и темпоральной логике действий (англ. TLA, temporal logic of actions).

Темпоральная логика была введена Амиром Пнуэли в 1970-х годах [11]. Лесли Лэмпорт увидел недостаточность этой идеи для описания систем целиком и пришёл к мысли о необходимости использовать конечные автоматы, которым придавался смысл формул темпоральной логики, описывающих все возможные пути исполнения. Таким образом родилась идея темпоральной логики действий (TLA), которая дополнила темпоральную логику следующим [12]:

- инвариантность при повторении состояния;
- темпоральное использование кванторов существования;
- принятие в качестве атомарных формул не только предикатов состояния, но и формул действий.

TLA-спецификация — темпоральная формула, часто называемая *Spec* и являющаяся предикатом (утверждением) о поведении. Поведение представляет собой возможный путь исполнения системы [13].

Состоянием называется присваивание значений переменных, шагом называется пара состояний. Теперь поведение можно представить как бесконечную последовательность состояний, а шагами поведения можно назвать пару последовательных состояний поведения. Предикатом состояния называется функция, результат которой — логическое значение истина или ложь — соответствует утверждению о состоянии. Действием называется функция, имеющая смысл предиката над шагом. В этой функции участвуют как переменные первого шага, так и второго, которые обычно отмечаются штрихом [13].

$$Spec \triangleq Init \wedge \Box[Next]_{(v_1, v_2, \ldots, v_n)}$$

Здесь $Init$ — предикат состояния, $Next$ — действие, $v_i$ — переменные, $\Box$ — единственный в данной спецификации темпоральный оператор (истинно во всех будущих состояниях).

TLC — это программа, которая по заданному TLA+ описанию системы и формулам свойств перебирает состояния системы и определяет, удовлетворяет ли система заданным свойствам.

Обычно работа с TLA+/TLC строится таким образом: описываем систему в TLA+, формализуем в TLA+ интересные свойства, запускаем TLC для проверки. Таким образом и будет строиться работа по описанию алгоритма консенсуса Raft.

## 4.1 Формальная спецификация алгоритма

Ниже приведён подробный разбор TLA+ спецификации алгоритма консенсуса Raft. Текст разделён на логические блоки (константы, переменные, вспомогательные функции, действия, спецификация и т.д.) в том порядке, в каком они представлены в коде. Полный код спецификации приведен в Приложении А.

### Константы

На рис. 5 описаны константы спецификации. Константы в TLA+ представляют собой неизменные в пределах спецификации значения . Они определяются в начале спецификации TLA+ и не изменяют свои значения в ходе выполнения модели системы. Константы используются для настройки параметров или значений конфигурации, которые определяют поведение системы, но не изменяются при изменении состояния системы. Они будут заданы в части "Проверка модели".

```
---------------------------------- MODULE Raft ----------------------------------

EXTENDS Naturals, FiniteSets, Sequences, TLC

\* The set of server IDs
CONSTANTS Server
\* The set of requests that can go into the log
CONSTANTS Value
\* Server states.
CONSTANTS Follower, Candidate, Leader
\* A reserved value.
CONSTANTS Nil
\* Message types:
CONSTANTS RequestVoteRequest, RequestVoteResponse,
          AppendEntriesRequest, AppendEntriesResponse
\* Used for filtering messages under different circumstance
CONSTANTS EqualTerm, LessOrEqualTerm
\* Limiting state space by limiting the number of elections and restarts
CONSTANTS MaxElections, MaxRestarts
```

Рисунок 5 – Описание констант спецификации

На рис. 5 описана «основная среда» протокола: какие есть сервера, значения, роли (Follower/Candidate/Leader), типы сообщений, специальные маркеры вроде Nil и ограничения для упрощения моделирования.

$EXTENDS$ перечисляет, какие стандартные модули TLA+ были подключены:

- $Naturals$: для работы с натуральными числами и базовыми операциями;
- $FiniteSets$: предоставляет операции над конечными множествами (например, $SUBSET(S)$);
- $Sequences$: даёт операции над последовательностями (например, $Len$, $Append$, $SubSeq$ и т.д.);
- $TLC$: вспомогательные средства моделирования (такие как $Permutations$, $Print$ и пр.), а также операторы типа $CHOOSE$.

$CONSTANTS\ Server$ - множество идентификаторов серверов. Традиционно, когда описывается система, состоящая из набора процессов/узлов, в TLA+ они задаются как константа-множество (например, $\{s1, s2, s3\}$).

$CONSTANTS\ Value$ – множество возможных «значений» (запросов), которые могут добавляться в журналы (logs) серверов.

CONSTANTS Follower, Candidate, Leader – три константы, обозначающие состояния серверов в Raft: Последователь (Follower), Кандидат (Candidate) и Лидер (Leader).

$CONSTANTS\ Nil$ – специальное «пустое» значение. Применяется в случаях, когда нужно указать «отсутствие чего-либо».

Блок $CONSTANTS$ $RequestVoteRequest$, $RequestVoteResponse$, $AppendEntriesRequest$, $AppendEntriesResponse$ – четыре типа сообщений в Raft: запрос на голосование, ответ на запрос голосования, запрос на добавления записей в журнал и ответ на добавление соответственно.

CONSTANTS EqualTerm, LessOrEqualTerm – два способа проверки периода (терма), которые используются при обработке сообщений: равенство периодов (EqualTerm) или меньшее/равное (LessOrEqualTerm).

CONSTANTS MaxElections, MaxRestarts – ограничения для ограничения пространства состояний (количество возможных выборов лидера и рестартов узлов).

### Глобальные переменные

Глобальная переменная (разделяемая между всеми серверами) только одна: $messages$. Она представляет собой «мультимножество» сообщений, которые циркулируют в системе. В коде это представлено как функция вида $Message \to Nat$, где $Nat$ – сколько раз сообщение встречается (сколько копий есть в канале).

### Вспомогательные переменные

На рис. 6 представлены вспомогательные переменные.

```
\* Auxilliary variables (used for state-space control, invariants etc)

\* The values that have been received from a client and whether
\* the value has been acked back to the client. Used in invariants to
\* detect data loss.
VARIABLE acked
\* Counter for elections and restarts (to control state space)
VARIABLE electionCtr, restartCtr
auxVars == <<acked, electionCtr, restartCtr>>
```

Рисунок 6 – Описание вспомогательных переменных

*acked* - это отображение вида $Value \to \{Nil, FALSE, TRUE\}$. Оно показывает, было ли значение «подтверждено» (acked) сервером обратно клиенту. Если $acked[v] = FALSE$, то запрос клиента уже поступил, но ещё не подтверждён; если $TRUE$, то подтверждён. Если $Nil$, значит, это значение пока вообще не предлагалось. Переменная используется в инвариантах для проверки отсутствия потери данных.

*electionCtr* и *restartCtr* – счётчики, ограничивающие пространство состояний. Определяют число выборов и рестартов, их максимальное значение задано константами $MaxElections$, $MaxRestarts$, которые были описаны выше.

### Переменные сервера

Все переменные, приведенные на рис. 7 определены для каждого сервера. Т.е. каждая из них - функция от идентификатора сервера, задаваемого констатами в $Server$.

```
\* The server's term number.
VARIABLE currentTerm
\* The server's state (Follower, Candidate, or Leader).
VARIABLE state
\* The candidate the server voted for in its current term, or
\* Nil if it hasn't voted for any.
VARIABLE votedFor
serverVars == <<currentTerm, state, votedFor>>

\* A Sequence of log entries. The index into this sequence is the index of the
\* log entry. Unfortunately, the Sequence module defines Head(s) as the entry
\* with index 1, so be careful not to use that!
VARIABLE log
\* The index of the latest entry in the log the state machine may apply.
VARIABLE commitIndex
logVars == <<log, commitIndex>>

\* The following variables are used only on candidates:

\* The set of servers from which the candidate has received a vote in its
\* currentTerm.
VARIABLE votesGranted

candidateVars == <<votesGranted>>

\* The following variables are used only on leaders:
\* The next entry to send to each follower.
VARIABLE nextIndex
\* The latest entry that each follower has acknowledged is the same as the
\* leader's. This is used to calculate commitIndex on the leader.
VARIABLE matchIndex
\* Tracks which peers a leader is waiting on a response for.
\* Used for one-at-a-time AppendEntries RPCs. Not really required but
\* permitting out of order requests explodes the state space.
VARIABLE pendingResponse
leaderVars == <<nextIndex, matchIndex, pendingResponse>>

\* End of per server variables.
```

Рисунок 7 – Переменные сервера

*currentTerm* – для каждого сервера i хранит номер (терм) текущего периода Raft. *state* – текущее состояние сервера: $Follower$, $Candidate$ или $Leader$, как было указано в константах.

*votedFor* - для каждого сервера хранит, кому он отдал голос в текущем терме (или Nil, если ещё не голосовал). Все они сгруппированы в переменную $serverVars$ для удобства обращения (например, в $WF\_vars(Next)$).

*log* - функция от $Server$, где каждому серверу ставится в соответствие последовательность ($Sequence$) записей в журнале. Элемент журнала обычно содержит $(term, value)$.

$commitIndex$ показывает, до какого индекса журнал считается подтвержденнным (т.е. готовым к применению к машине состояний).

$votesGranted$ – для сервера, который находится в состоянии $Candidate$, здесь хранится множество серверов, которые этому кандидату дали голос в текущем терме.

$nextIndex$ – для каждого лидера и для каждого сервера в кластере хранит индекс, с которого лидер должен отправлять записи в лог для репликации. То есть, если $nextIndex[i][j] = k$, значит лидер $i$ собирается отправить серверу $j$ записи, начиная с индекса $k$ (в логе самого лидера).

$matchIndex$ – для лидера $i$ и сервера $j$ указывает, до какого индекса журнала сервер $j$ гарантированно имеет те же записи, что и лидер.

$pendingResponse$ – для лидера $i$ и сервера $j$ это булево значение, показывающее, ждет ли лидер ответа на предыдущий RPC. Если $TRUE$, значит лидер уже отправил запрос $AppendEntries$ серверу $j$ и ещё не получил ответа; чтобы в модели не плодилось слишком много состояний, запрещено отправлять параллельные $AppendEntries$ одному и тому же серверу.

### Вспомогательные функции

Реализация всех вспомогательных функций представлена на рис. 8. Они не используются в инициализации системы, пересылке или обработке сообщений.

$Quorum$ – множество всех подмножеств серверов, которые образуют мажоритарный кворум (подмножество, размер которого более половины). Если размер кластера $N$, то в кворуме строго более $N/2$ узлов. Каждый кворум пересекается с любым другим кворумом хотя бы в одном узле.

$LastTerm(xlog)$ – возвращает терм последней записи в журнале, или 0, если журнал пуст.

$\_SendNoRestriction(m)$ – отправить сообщение $m$, увеличив счётчик копий в $messages$. Если сообщение уже есть, увеличим счётчик, если нет – добавим новую запись $m :> 1$.

```
Quorum == {i \in SUBSET(Server) : Cardinality(i) * 2 > Cardinality(Server)}

LastTerm(xlog) == IF Len(xlog) = 0 THEN 0 ELSE xlog[Len(xlog)].term

_SendNoRestriction(m) ==
    IF m \in DOMAIN messages
    THEN messages' = [messages EXCEPT ![m] = @ + 1]
    ELSE messages' = messages @@ (m :> 1)

\* Will only send the message if it hasn't been sent before.
\* Basically disables the parent action once sent.
_SendOnce(m) ==
    /\ m \notin DOMAIN messages
    /\ messages' = messages @@ (m :> 1)

\* Add a message to the bag of messages.
Send(m) ==
    IF /\ m.mtype = AppendEntriesRequest
       /\ m.mentries = <<>>
    THEN _SendOnce(m)
    ELSE _SendNoRestriction(m)

\* Will only send the messages if it hasn't done so before
SendMultipleOnce(msgs) ==
    /\ \A m \in msgs : m \notin DOMAIN messages
    /\ messages' = messages @@ [msg \in msgs |-> 1]

\* Explicit duplicate operator for when we purposefully want message duplication
Duplicate(m) ==
    /\ m \in DOMAIN messages
    /\ messages' = [messages EXCEPT ![m] = @ + 1]

\* Remove a message from the bag of messages. Used when a server is done
\* processing a message.
Discard(m) ==
    /\ m \in DOMAIN messages
    /\ messages[m] > 0 \* message must exist
    /\ messages' = [messages EXCEPT ![m] = @ - 1]

\* Combination of Send and Discard
Reply(response, request) ==
    /\ messages[request] > 0 \* request must exist
    /\ \/ /\ response \notin DOMAIN messages \* response does not exist, so add it
          /\ messages' = [messages EXCEPT ![request] = @ - 1] @@ (response :> 1)
       \/ /\ response \in DOMAIN messages \* response was sent previously, so increment delivery counter
          /\ messages' = [messages EXCEPT ![request] = @ - 1,
                                          ![response] = @ + 1]

\* The message is of the type and has a matching term.
ReceivableMessage(m, mtype, term_match) ==
    /\ messages[m] > 0
    /\ m.mtype = mtype
    /\ \/ /\ term_match = EqualTerm
          /\ m.mterm = currentTerm[m.mdest]
       \/ /\ term_match = LessOrEqualTerm
          /\ m.mterm <= currentTerm[m.mdest]

\* Return the minimum value from a set, or undefined if the set is empty.
Min(s) == CHOOSE x \in s : \A y \in s : x <= y
\* Return the maximum value from a set, or undefined if the set is empty.
Max(s) == CHOOSE x \in s : \A y \in s : x >= y
```

Рисунок 8 – Вспомогательные функции

$\_SendOnce(m)$ – отправить сообщение m только один раз. Действие разрешено только если такого сообщения ещё нет в $messages$. Это используется для тех случаев, когда повторная отправка может сильно увеличивать пространство состояний (например, пустые RPC не должны слаться бесконечно).

$Send(m)$ – оператор отправки сообщения. Если это $AppendEntriesRequest$ без данных ($mentries$ $=<<>>$), то используем правило «послать только один раз»; иначе же можно добавлять в мешок сколько угодно раз.

$SendMultipleOnce(msgs)$ – отправляем сразу несколько сообщений за один шаг, но только если ни одно из них ранее не присутствовало. Множество $msgs$ – набор сообщений (часто это рассылка $RequestVote$ всем остальным).

$Duplicate(m)$ – искусственно дублирует сообщение, увеличивая его счётчик. Это отдельное действие «сеть может продублировать сообщение».

$Discard(m)$ – «убрать одно вхождение сообщения из мешка». Если там несколько копий, счётчик уменьшится на 1. Обычно вызывается после обработки сообщения сервером.

$Reply(response, request)$ – соединяет операцию «удалить входящее сообщение ($request$)» и «положить/увеличить ответ (response)» за один шаг. Если ответ уже существует, увеличиваем счётчик; если нет – добавляем.

$Min/Max$ – функции для выборки минимального/максимального элемента из множества.

### Инициализация

На рис. 9 представлена инициализация модели, по выполнении которой формируется полное начальное состояние Raft-кластера: все сервера Follower'ы с термом 1, без записей в журнале, без активных сообщений и с нулевыми счётчиками.

Важно отметить, что все переменные, кроме $electionCtr$, $restartCtr$, $acked$ и $messages$ объявляются как функция от $Server$ (i in Server |->), можно воспринимать их также как таблицу $map$.

```
InitServerVars == /\ currentTerm = [i \in Server |-> 1]
                  /\ state       = [i \in Server |-> Follower]
                  /\ votedFor    = [i \in Server |-> Nil]
InitCandidateVars == votesGranted   = [i \in Server |-> {}]
\* The values nextIndex[i][i] and matchIndex[i][i] are never read, since the
\* leader does not send itself messages. It's still easier to include these
\* in the functions.
InitLeaderVars == /\ nextIndex  = [i \in Server |-> [j \in Server |-> 1]]
                  /\ matchIndex = [i \in Server |-> [j \in Server |-> 0]]
InitLogVars == /\ log             = [i \in Server |-> << >>]
               /\ commitIndex     = [i \in Server |-> 0]
               /\ pendingResponse = [i \in Server |-> [j \in Server |-> FALSE]]
InitAuxVars == /\ electionCtr = 0
               /\ restartCtr = 0
               /\ acked = [v \in Value |-> Nil]

Init == /\ messages = [m \in {} |-> 0]
        /\ InitServerVars
        /\ InitCandidateVars
        /\ InitLeaderVars
        /\ InitLogVars
        /\ InitAuxVars
```

Рисунок 9 – Инициализация модели

### Действия

В Raft есть несколько ключевых переходов. Каждый переход описывает, при каких условиях он выполняется и как меняет состояние. Затем все они объединяются в оператор $Next$, который говорит: «возможен любой из этих переходов».

**Restart**

Код метода *Restart* представлен на рис. 10. Он переводит сервер в состояние $Follower$, сбрасывает все «volatile» переменные ($votesGranted$, $nextIndex$, $matchIndex$, $pendingResponse$, $commitIndex$), увеличивает счётчик $restartCtr$. При этом не изменяются «persistent» переменные: терм ($currentTerm[i]$), $votedFor[i]$, сам журнал ($log[i]$), уже подтверждённые значения. Условие $restartCtr < MaxRestarts$ ограничивает количество рестартов.

```
Restart(i) ==
    /\ restartCtr < MaxRestarts
    /\ state'          = [state EXCEPT ![i] = Follower]
    /\ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
    /\ nextIndex'      = [nextIndex EXCEPT ![i] = [j \in Server |-> 1]]
    /\ matchIndex'     = [matchIndex EXCEPT ![i] = [j \in Server |-> 0]]
    /\ pendingResponse' = [pendingResponse EXCEPT ![i] = [j \in Server |-> FALSE]]
    /\ commitIndex'    = [commitIndex EXCEPT ![i] = 0]
    /\ restartCtr'     = restartCtr + 1
    /\ UNCHANGED <<messages, currentTerm, votedFor, log, acked, electionCtr>>
```

Рисунок 10 – Метод *Restart*

**RequestVote**

Код метода $RequestVote$ представлен на рис. 11. Сервер $i$ будучи в состоянии $Follower$ или $Candidate$, начинает новые выборы следующим образом:

а) Проверка, что ещё не превышен $MaxElections$;

б) Состояние меняется в $Candidate$;

в) Терм увеличивается на 1;

г) Отдается голос самому себе: $votedFor[i] = i$;

д) Увеличивается счетчик выборов;

е) Рассылаются запросы на голос всем серверам, кроме себя ($RequestVoteRequest$).

```
RequestVote(i) ==
    /\ electionCtr < MaxElections
    /\ state[i] \in {Follower, Candidate}
    /\ state' = [state EXCEPT ![i] = Candidate]
    /\ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
    /\ votedFor' = [votedFor EXCEPT ![i] = i] \* votes for itself
    /\ votesGranted'   = [votesGranted EXCEPT ![i] = {i}] \* votes for itself
    /\ electionCtr' = electionCtr + 1
    /\ SendMultipleOnce(
          {[mtype         |-> RequestVoteRequest,
            mterm         |-> currentTerm[i] + 1,
            mlastLogTerm  |-> LastTerm(log[i]),
            mlastLogIndex |-> Len(log[i]),
            msource       |-> i,
            mdest         |-> j] : j \in Server \ {i}})
    /\ UNCHANGED <<acked, leaderVars, logVars, restartCtr>>
```

Рисунок 11 – Метод $RequestVote$

**AppendEntries**

Код метода $AppendEntries$ представлен на рис. 12.

```
AppendEntries(i, j) ==
    /\ i /= j
    /\ state[i] = Leader
    /\ pendingResponse[i][j] = FALSE \* not already waiting for a response
    /\ LET prevLogIndex == nextIndex[i][j] - 1
           prevLogTerm == IF prevLogIndex > 0 THEN
                             log[i][prevLogIndex].term
                          ELSE
                             0
           \* Send up to 1 entry, constrained by the end of the log.
           lastEntry == Min({Len(log[i]), nextIndex[i][j]})
           entries == SubSeq(log[i], nextIndex[i][j], lastEntry)
       IN
          /\ pendingResponse' = [pendingResponse EXCEPT ![i][j] = TRUE]
          /\ Send([mtype          |-> AppendEntriesRequest,
                   mterm          |-> currentTerm[i],
                   mprevLogIndex  |-> prevLogIndex,
                   mprevLogTerm   |-> prevLogTerm,
                   mentries       |-> entries,
                   mcommitIndex   |-> Min({commitIndex[i], lastEntry}),
                   msource        |-> i,
                   mdest          |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, nextIndex, matchIndex, logVars, auxVars>>
```

Рисунок 12 – Метод $AppendEntries$

Лидер $i$ отправляет запрос *AppendEntriesRequest* последователю $j$. В сообщении не может быть больше одной записи *entry*. Условие pendingResponse[i][j] = FALSE гарантирует, что не отправляем новый RPC, пока старый еще без ответа.

**BecomeLeader**

Код метода *BecomeLeader* представлен на рис. 13. Сервер-кандидат $i$ обнаруживает, что у него есть кворум голосов (в $votesGranted[i]$). Значит, он выигрывает выборы и становится лидером. Переводится $state[i]$ в *Leader*, инициализируется $nextIndex[i]$, $matchIndex[i]$, $pendingResponse[i]$. Другие переменные не меняются.

```
BecomeLeader(i) ==
    /\ state[i] = Candidate
    /\ votesGranted[i] \in Quorum
    /\ state'      = [state EXCEPT ![i] = Leader]
    /\ nextIndex'  = [nextIndex EXCEPT ![i] =
                         [j \in Server |-> Len(log[i]) + 1]]
    /\ matchIndex' = [matchIndex EXCEPT ![i] =
                         [j \in Server |-> 0]]
    /\ pendingResponse' = [pendingResponse EXCEPT ![i] =
                               [j \in Server |-> FALSE]]
    /\ UNCHANGED <<messages, currentTerm, votedFor, candidateVars,
                   auxVars, logVars>>
```

Рисунок 13 – Метод *BecomeLeader*

**AdvanceCommitIndex**

Код метода *AdvanceCommitIndex* представлен на рис. 14.

```
AdvanceCommitIndex(i) ==
    /\ state[i] = Leader
    /\ LET \* The set of servers that agree up through index.
           Agree(index) == {i} \cup {k \in Server :
                                     /\ matchIndex[i][k] >= index }
           \* The maximum indexes for which a quorum agrees
           agreeIndexes == {index \in 1..Len(log[i]) :
                               Agree(index) \in Quorum}
           \* New value for commitIndex'[i]
           newCommitIndex ==
              IF /\ agreeIndexes /= {}
                 /\ log[i][Max(agreeIndexes)].term = currentTerm[i]
              THEN
                  Max(agreeIndexes)
              ELSE
                  commitIndex[i]
       IN
          /\ commitIndex[i] < newCommitIndex \* only enabled if it actually advances
          /\ commitIndex' = [commitIndex EXCEPT ![i] = newCommitIndex]
          /\ acked' = [v \in Value |->
                        IF acked[v] = FALSE
                        THEN v \in { log[i][index].value : index \in commitIndex[i]+1..newCommitIndex }
                        ELSE acked[v]]
    /\ UNCHANGED <<messages, serverVars, candidateVars, leaderVars, log,
                   pendingResponse, electionCtr, restartCtr>>
```

Рисунок 14 – Метод *AdvanceCommitIndex*

Лидер $i$ пытается продвинуть свой $commitIndex$ вперёд, если есть подтверждение большинства узлов для определённого индекса. Для этого проверяется множество серверов, у которых $matchIndex[i][k] >= index$, и вычисляется максимум индексов, по которым у лидера есть кворум. Если этот индекс принадлежит текущему терму лидера, обновляется $commitIndex[i]$. Все значения, которые таким образом становятся закоммиченными, теперь помечаются $acked[v] = TRUE$.

**UpdateTerm**

Код метода $UpdateTerm$ представлен на рис. 15. Если где-то в сети есть сообщение с термом больше, чем у сервера, который его получит, то сервер «обновляет» свой терм, сбрасывает своё голосование, становится $Follower$.

```
UpdateTerm ==
    \E m \in DOMAIN messages :
        /\ m.mterm > currentTerm[m.mdest]
        /\ currentTerm'    = [currentTerm EXCEPT ![m.mdest] = m.mterm]
        /\ state'          = [state       EXCEPT ![m.mdest] = Follower]
        /\ votedFor'       = [votedFor    EXCEPT ![m.mdest] = Nil]
           \* messages is unchanged so m can be processed further.
        /\ UNCHANGED <<messages, candidateVars, leaderVars, logVars, auxVars>>
```

Рисунок 15 – Метод $UpdateTerm$

**HandleRequestVoteRequest**

Код метода $HandleRequestVoteRequest$ представлен на рис. 16.

```
HandleRequestVoteRequest ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, RequestVoteRequest, LessOrEqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == \/ m.mlastLogTerm > LastTerm(log[i])
                        \/ /\ m.mlastLogTerm = LastTerm(log[i])
                           /\ m.mlastLogIndex >= Len(log[i])
               grant == /\ m.mterm = currentTerm[i]
                        /\ logOk
                        /\ votedFor[i] \in {Nil, j}
           IN /\ m.mterm <= currentTerm[i]
              /\ \/ grant  /\ votedFor' = [votedFor EXCEPT ![i] = j]
                 \/ ~grant /\ UNCHANGED votedFor
              /\ Reply([mtype        |-> RequestVoteResponse,
                        mterm        |-> currentTerm[i],
                        mvoteGranted |-> grant,
                        msource      |-> i,
                        mdest        |-> j],
                        m)
              /\ UNCHANGED <<state, currentTerm, candidateVars, leaderVars,
                             logVars, auxVars>>
```

Рисунок 16 – Метод $HandleRequestVoteRequest$

Сервер $i$ обрабатывает входящий $RequestVoteRequest$ от сервера $j$. Сначала проверяется условие *ReceivableMessage(..., LessOrEqualTerm)*, то есть терм запроса не больше текущего, иначе этим занялось бы $UpdateTerm$. Затем проверяется «лог последнего кандидата»: *(m.mlastLogTerm, m.mlastLogIndex)*

сравнивается с собственным журналом. Если лог кандидата не меньше нашего, и сам сервер $i$ ещё не голосовал в этом терме (или уже голосовал за $j$), мы отдадим ему голос. Результат — отправка $RequestVoteResponse$ с $mvoteGranted = grant$.

**HandleRequestVoteResponse**

Код метода $HandleRequestVoteResponse$ представлен на рис. 17. Сервер $i$ получает ответ на запрос голосования от $j$. Если $mvoteGranted = TRUE$, добавляем $j$ в $votesGranted[i]$. Иначе ничего не делаем. Сообщение удаляется из сети после обработки.

```
HandleRequestVoteResponse ==
    \E m \in DOMAIN messages :
        \* This tallies votes even when the current state is not Candidate, but
        \* they won't be looked at, so it doesn't matter.
        /\ ReceivableMessage(m, RequestVoteResponse, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
           IN
              /\ \/ /\ m.mvoteGranted
                    /\ votesGranted' = [votesGranted EXCEPT ![i] =
                                            votesGranted[i] \cup {j}]
                 \/ /\ ~m.mvoteGranted
                    /\ UNCHANGED <<votesGranted>>
              /\ Discard(m)
              /\ UNCHANGED <<serverVars, votedFor, leaderVars, logVars,
                             auxVars>>
```

Рисунок 17 – Метод $HandleRequestVoteResponse$

**RejectAppendEntriesRequest**

Код метода $RejectAppendEntriesRequest$ представлен на рис. 18. Сервер $i$ отвергает $AppendEntriesRequest$, если терм сообщения меньше его собственного либо если термы равны, но у получателя – $Follower$, и при этом журнал «не совпадает» (поле $m.mprevLogTerm$ или $m.mprevLogIndex$ не соответствуют локальному журналу).

```
RejectAppendEntriesRequest ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, AppendEntriesRequest, LessOrEqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == LogOk(i, m)
           IN  /\ \/ m.mterm < currentTerm[i]
                  \/ /\ m.mterm = currentTerm[i]
                     /\ state[i] = Follower
                     /\ \lnot logOk
               /\ Reply([mtype           |-> AppendEntriesResponse,
                         mterm           |-> currentTerm[i],
                         msuccess        |-> FALSE,
                         mmatchIndex     |-> 0,
                         msource         |-> i,
                         mdest           |-> j],
                         m)
               /\ UNCHANGED <<state, candidateVars, leaderVars, serverVars,
                              logVars, auxVars>>
```

Рисунок 18 – Метод $RejectAppendEntriesRequest$

В ответ отправляется *AppendEntriesResponse* с $msuccess = FALSE$.

**AcceptAppendEntriesRequest**

Код метода *AcceptAppendEntriesRequest* представлен на рис. 19. Может произойти урезание лога и/или добавление одной записи. $commitIndex'[i]$ устанавливается равным тому, что лидер прислал в $m.mcommitIndex$, так как *Follower* следует за лидером. Отправляем положительный ответ (*AppendEntriesResponse*) с $msuccess = TRUE$.

```
AcceptAppendEntriesRequest ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, AppendEntriesRequest, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == LogOk(i, m)
               index == m.mprevLogIndex + 1
           IN
              /\ state[i] \in { Follower, Candidate }
              /\ logOk
              /\ LET new_log == CASE CanAppend(m, i) ->
                                       [log EXCEPT ![i] = Append(log[i], m.mentries[1])]
                                 [] NeedsTruncation(m, i , index) /\ m.mentries # <<>> ->
                                       [log EXCEPT ![i] = Append(TruncateLog(m, i), m.mentries[1])]
                                 [] NeedsTruncation(m, i , index) /\ m.mentries = <<>> ->
                                       [log EXCEPT ![i] = TruncateLog(m, i)]
                                 [] OTHER -> log
                 IN
                    /\ state' = [state EXCEPT ![i] = Follower]
                    /\ commitIndex' = [commitIndex EXCEPT ![i] =
                                              m.mcommitIndex]
                    /\ log' = new_log
                    /\ Reply([mtype           |-> AppendEntriesResponse,
                              mterm           |-> currentTerm[i],
                              msuccess        |-> TRUE,
                              mmatchIndex     |-> m.mprevLogIndex +
                                                  Len(m.mentries),
                              msource         |-> i,
                              mdest           |-> j],
                              m)
                    /\ UNCHANGED <<candidateVars, leaderVars, votedFor, currentTerm,
                                   auxVars>>
```

Рисунок 19 – Метод *AcceptAppendEntriesRequest*

**HandleAppendEntriesResponse**

Код метода *HandleAppendEntriesResponse* представлен на рис. **??**.

```
HandleAppendEntriesResponse ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, AppendEntriesResponse, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
           IN
              /\ \/ /\ m.msuccess \* successful
                    /\ nextIndex'  = [nextIndex  EXCEPT ![i][j] = m.mmatchIndex + 1]
                    /\ matchIndex' = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
                 \/ /\ \lnot m.msuccess \* not successful
                    /\ nextIndex' = [nextIndex EXCEPT ![i][j] =
                                          Max({nextIndex[i][j] - 1, 1})]
                    /\ UNCHANGED <<matchIndex>>
              /\ pendingResponse' = [pendingResponse EXCEPT ![i][j] = FALSE]
              /\ Discard(m)
              /\ UNCHANGED <<serverVars, candidateVars, logVars, auxVars>>
```

Рисунок 20 – Метод *HandleAppendEntriesResponse*

Лидер $i$ получает ответ от $j$. Если $m.msuccess = TRUE$, значит $j$ у себя добавил записи, и лидер ставит $matchIndex[i][j] = mmatchIndex$. $nextIndex[i][j]$ становится $mmatchIndex + 1$. Если $FALSE$, значит у $j$ оказались расхождения, и тогда лидер уменьшает $nextIndex[i][j]$ на 1, чтобы попробовать снова. $pendingResponse'[i][j] = FALSE$ – освобождается «окно» для следующего запроса. Затем сообщение выбрасывается из сети.

### Спецификация и оператор Next

Код оператора и спецификации приведен на рис. 21

```
\* Defines how the variables may transition.
Next ==
        \/ \E i \in Server : Restart(i)
        \/ \E i \in Server : RequestVote(i)
        \/ \E i \in Server : BecomeLeader(i)
        \/ \E i \in Server : AdvanceCommitIndex(i)
        \/ \E i,j \in Server : AppendEntries(i, j)
        \/ UpdateTerm
        \/ HandleRequestVoteRequest
        \/ HandleRequestVoteResponse
        \/ RejectAppendEntriesRequest
        \/ AcceptAppendEntriesRequest
        \/ HandleAppendEntriesResponse

\* The specification must start with the initial state and transition according
\* to Next.
NoStuttering ==
    WF_vars(Next)

Spec == Init /\ [][Next]_vars

LivenessSpec == Init /\ [][Next]_vars /\ NoStuttering
```

Рисунок 21 – Спецификация и оператор $Next$

Оператор $Next$ представляет собой главную дизъюнкцию всех возможных действий, которые могут происходить в любой момент. Система в каждый момент времени может сделать один из перечисленных переходов (*Restart, RequestVote, AppendEntries, BecomeLeader* и т.п.), если выполняются условия.

*Spec* – классическая структура TLA+ спецификации: $Init$ задаёт начальное состояние, $[][Next]\_vars$ говорит, что на каждом шаге выполняется одно из действий $Next$.

### Инварианты

Инварианты представлены на рис. 22. Они проверяются моделью, описанной в следующей части, чтобы убедиться, что Raft-свойства не нарушаются при любых последовательностях действий.

```
MinCommitIndex(s1, s2) ==
    IF commitIndex[s1] < commitIndex[s2]
    THEN commitIndex[s1]
    ELSE commitIndex[s2]

\* INV: NoLogDivergence
\* The log index is consistent across all servers (on those
\* servers whose commitIndex is equal or higher than the index).
NoLogDivergence ==
    \A s1, s2 \in Server :
        IF s1 = s2
        THEN TRUE
        ELSE
            LET lowest_common_ci == MinCommitIndex(s1, s2)
            IN IF lowest_common_ci > 0
               THEN \A index \in 1..lowest_common_ci : log[s1][index] = log[s2][index]
               ELSE TRUE

\* INV: LeaderHasAllAckedValues
\* A non-stale leader cannot be missing an acknowledged value
LeaderHasAllAckedValues ==
    \* for every acknowledged value
    \A v \in Value :
        IF acked[v] = TRUE
        THEN
            \* there does not exist a server that
            ~\E i \in Server :
                \* is a leader
                /\ state[i] = Leader
                \* and which is the newest leader (aka not stale)
                /\ ~\E l \in Server :
                    /\ l # i
                    /\ currentTerm[l] > currentTerm[i]
                \* and that is missing the value
                /\ ~\E index \in DOMAIN log[i] :
                    log[i][index].value = v
        ELSE TRUE
```

Рисунок 22 – Инварианты

*NoLogDivergence* – инвариант о том, что если у двух серверов некий индекс подтвержден (меньшее из их commitIndex), то записи по этому индексу совпадают. Это ключевое свойство согласованности в Raft.

$LeaderHasAllAckedValues$ – если значение ($v$) уже подтверждено как $acked[v] = TRUE$, то любой лидер обязан иметь это значение в своём логе.

**Итого**

Данная спецификация описывает основные аспекты Raft:

а) Состояния узлов (*Follower, Candidate, Leader*) и их персистентные переменные (*currentTerm, votedFor, log*).

б) Обмен сообщениями (*RequestVote, AppendEntries* и их ответы) через переменную *messages*.

в) Логика перехода (кандидат пытается стать лидером, лидер реплицирует записи, узлы отвечают и обновляют термы, если видят более высокий терм, и т.д.).

г) Инварианты (*NoLogDivergence, LeaderHasAllAckedValues*, и пр.), которые отражают ключевые свойства безопасности Raft.

Таким образом, в модели учтены основные механизмы Raft: выбор лидера, поддержка согласованного журнала, коммиты и гарантии о том, что система не теряет записанные и закоммиченные данные.

### 4.2 Проверка модели

В TLA+ модель проверяется с помощью TLC, перебирающего всевозможные (или ограниченные) пути выполнения спецификации и проверяющего инварианты.

В приложении Б приведен код конфигурации модели для TLC, в котором задаются значения констант, какие варианты проврять, какой оператор считать стартовым и переходным.

Результаты проверки приведены на рис. 23. Было проверено более 36 тысяч возможных состояний, инварианты не нарушаются.

```
>  tlc -workers 6 -deadlock raft
TLC2 Version 2.18 of Day Month 20?? (rev: 5b3286b)
Running breadth-first search Model-Checking with fp 24 and seed -5005896700769763715 with 6 workers on 6
    cores with 7116MB heap and 64MB offheap memory [pid: 167904] (Linux 6.12.8 amd64, N/A 21.0.5 x86_64,
    MSBDiskFPSet, DiskStateQueue).
Parsing file /home/serpentian/tla/raft.tla
Parsing file /tmp/tlc-14441480457504028982/Naturals.tla (jar:file:/nix/store/
    k6znsd1l1w932fjh7p61b76bp6jahwmn-tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/
    Naturals.tla)59jv-texlive-combined-full-2024.20
Parsing file /tmp/tlc-14441480457504028982/FiniteSets.tla (jar:file:/nix/store/
    k6znsd1l1w932fjh7p61b76bp6jahwmn-tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/
    FiniteSets.tla)
Parsing file /tmp/tlc-14441480457504028982/Sequences.tla (jar:file:/nix/store/
    k6znsd1l1w932fjh7p61b76bp6jahwmn-tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/
    Sequences.tla)
Parsing file /tmp/tlc-14441480457504028982/TLC.tla (jar:file:/nix/store/k6znsd1l1w932fjh7p61b76bp6jahwmn-
    tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/TLC.tla)          59jv-texlive-
    combined-full-2024.20Parsing file /tmp/tlc-14441480457504028982/_TLCTrace.tla (jar:file:/nix/store/
    k6znsd1l1w932fjh7p61b76bp6jahwmn-tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/
    _TLCTrace.tla)
Parsing file /tmp/tlc-14441480457504028982/TLCExt.tla (jar:file:/nix/store/k6znsd1l1w932fjh7p61b76bp6jahwmn-
    tlaplus-1.8.0/share/java/tla2tools.jar!/tla2sany/StandardModules/TLCExt.tla)
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module FiniteSets
Semantic processing of module TLC
Semantic processing of module TLCExt
Semantic processing of module _TLCTrace
Semantic processing of module raft
Starting... (2025-01-19 18:16:01)
Computing initial states...
Finished computing initial states: 1 distinct state generated at 2025-01-19 18:16:01.
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
  calculated (optimistic):  val = 1.8E-10
  based on the actual fingerprints:  val = 1.3E-10
126718 states generated, 36538 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 29.
The average outdegree of the complete state graph is 1 (minimum is 0, the maximum 6 and the 95th percentile
    is 3).
Finished in 02s at (2025-01-19 18:16:03)
```

Рисунок 23 – Проверка модели

# ЗАКЛЮЧЕНИЕ

В ходе проведённой работы были проанализированы основные аспекты реализации консенсуса в распределённых системах с учётом заранее определённых ограничений (модель сети, уровни синхронности и типы отказов). Рассмотрены и сопоставлены два известных алгоритма консенсуса — Paxos и Raft: первый отличается сильной теоретической базой, однако достаточно сложен для понимания и практической имплементации; второй (Raft) спроектирован с упором на простоту объяснения и реализует те же гарантии согласованности.

В практической части работы был выбран алгоритм Raft и выполнена его формальная спецификация на языке TLA+. С помощью соответствующей конфигурации модели была осуществлена проверка корректности поведения в рамках заданных условий. Результаты верификации подтвердили соответствие системы основным свойствам безопасности (отсутствие расхождений в закоммиченных данных) и продемонстрировали правильную работу механизма выбора лидера и репликации лога.

Таким образом, проделанное исследование показало, что формальное описание и проверка протокола Raft с помощью TLA+ существенно повышают надёжность разработки и помогают выявлять потенциальные ошибки на ранних этапах проектирования. Полученные спецификации и результаты моделирования могут служить основой для дальнейшего расширения модели (например, учёта большего количества серверов, более сложных типов отказов) или для сравнения с другими алгоритмами консенсуса в рамках дальнейших исследований.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Таненбаум Э., Ван-Стеен М. Распределенные системы. Принципы и парадигмы. — Спб.: Питер, 2003. — С. 877.

2. Качин К., Гуерру Р., Родригес Л. Введение в надежное и безопасное распределенное программирование. — М.: ДМК Пресс, 2011. — С. 512.

3. Петров А. Распределенные данные. Алгоритмы работы современных систем хранения информации. — Спб.: Питер, 2024. — С. 336.

4. Fischer M. J., Lynch N. A., Paterson M. Impossibility of Distributed Consensus with One Faulty Process // J. ACM. — 1985. — Т. 32, № 2. — С. 374—382.

5. Skeen, Dale, and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. — 1983. — Т. SE—9, № 3. — С. 219—228.

6. Skeen, Dale, and M. Stonebraker. // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. — 1983. — Т. SE—9, № 3. — С. 219—228.

7. Lamport, Leslie. The part-time parliament // ACM Transactions on Computer Systems. — 1998. — Т. 16, № 2. — С. 133—169.

8. Lamport, Leslie. Paxos made simple // ACM SIGACT News. — 2001. — Т. 32, № 4. — С. 51—58.

9. Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. Paxos made live: an egnineering perspective // In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. — 2007. — С. 398—407.

10. Ongaro, Diego, and John Ousterhout. In search of an understandable consensus algorithm // In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference. — 2006. — С. 305—320.

11. Amir Pnueli. The temporal logic of programs // In Proceedings of the 18th Annual Symposium on the Foundations of Computer Science. — 1977. — С. 46—57.

12. Lamport L. The Specification Language TLA+. — 2008. — С. 616—620.

13. Henri Habrias, Marc Frappier. Software Specification Methods. — John Wiley and Sons, 2006. — С. 418.

# ПРИЛОЖЕНИЕ А

## Спецификация Raft на TLA+

Листинг А.1 – Реализация Raft

```
---------------------------------- MODULE Raft ---------------------------------

EXTENDS Naturals, FiniteSets, Sequences, TLC

CONSTANTS Server
CONSTANTS Value
CONSTANTS Follower, Candidate, Leader
CONSTANTS Nil
CONSTANTS RequestVoteRequest, RequestVoteResponse,
          AppendEntriesRequest, AppendEntriesResponse
CONSTANTS EqualTerm, LessOrEqualTerm
CONSTANTS MaxElections, MaxRestarts

----
\* Global variables
VARIABLE messages
VARIABLE acked
VARIABLE electionCtr, restartCtr
auxVars == <<acked, electionCtr, restartCtr>>
----
\* Per server variables (functions with domain Server).
VARIABLE currentTerm
VARIABLE state
VARIABLE votedFor
serverVars == <<currentTerm, state, votedFor>>

VARIABLE log
VARIABLE commitIndex
logVars == <<log, commitIndex>>

VARIABLE votesGranted
candidateVars == <<votesGranted>>

VARIABLE nextIndex
VARIABLE matchIndex
VARIABLE pendingResponse
leaderVars == <<nextIndex, matchIndex, pendingResponse>>
----
\* All variables; used for stuttering (asserting state hasn't changed).
vars == <<messages, serverVars, candidateVars, leaderVars, logVars, auxVars>>
view == <<messages, serverVars, candidateVars, leaderVars, logVars >>
symmServers == Permutations(Server)
symmValues == Permutations(Value)
----

\* Helpers
Quorum == {i \in SUBSET(Server) : Cardinality(i) * 2 > Cardinality(Server)}

LastTerm(xlog) == IF Len(xlog) = 0 THEN 0 ELSE xlog[Len(xlog)].term

_SendNoRestriction(m) ==
    IF m \in DOMAIN messages
    THEN messages' = [messages EXCEPT ![m] = @ + 1]
    ELSE messages' = messages @@ (m :> 1)

_SendOnce(m) ==
    /\ m \notin DOMAIN messages
    /\ messages' = messages @@ (m :> 1)

Send(m) ==
    IF /\ m.mtype = AppendEntriesRequest
```

```
       /\ m.mentries = <<>>
    THEN _SendOnce(m)
    ELSE _SendNoRestriction(m)

SendMultipleOnce(msgs) ==
    /\ \A m \in msgs : m \notin DOMAIN messages
    /\ messages' = messages @@ [msg \in msgs |-> 1]

Duplicate(m) ==
    /\ m \in DOMAIN messages
    /\ messages' = [messages EXCEPT ![m] = @ + 1]

Discard(m) ==
    /\ m \in DOMAIN messages
    /\ messages[m] > 0 \* message must exist
    /\ messages' = [messages EXCEPT ![m] = @ - 1]

Reply(response, request) ==
    /\ messages[request] > 0 \* request must exist
    /\ \/ /\ response \notin DOMAIN messages \* response does not exist, so add it
          /\ messages' = [messages EXCEPT ![request] = @ - 1] @@ (response :> 1)
       \/ /\ response \in DOMAIN messages \* response was sent previously, so increment delivery counter
          /\ messages' = [messages EXCEPT ![request] = @ - 1,
                                          ![response] = @ + 1]

ReceivableMessage(m, mtype, term_match) ==
    /\ messages[m] > 0
    /\ m.mtype = mtype
    /\ \/ /\ term_match = EqualTerm
          /\ m.mterm = currentTerm[m.mdest]
       \/ /\ term_match = LessOrEqualTerm
          /\ m.mterm <= currentTerm[m.mdest]

Min(s) == CHOOSE x \in s : \A y \in s : x <= y
Max(s) == CHOOSE x \in s : \A y \in s : x >= y

----
\* Define initial values for all variables
InitServerVars == /\ currentTerm = [i \in Server |-> 1]
                  /\ state       = [i \in Server |-> Follower]
                  /\ votedFor    = [i \in Server |-> Nil]
InitCandidateVars == votesGranted   = [i \in Server |-> {}]
InitLeaderVars == /\ nextIndex  = [i \in Server |-> [j \in Server |-> 1]]
                  /\ matchIndex = [i \in Server |-> [j \in Server |-> 0]]
InitLogVars == /\ log             = [i \in Server |-> << >>]
               /\ commitIndex     = [i \in Server |-> 0]
               /\ pendingResponse = [i \in Server |-> [j \in Server |-> FALSE]]
InitAuxVars == /\ electionCtr = 0
               /\ restartCtr = 0
               /\ acked = [v \in Value |-> Nil]

Init == /\ messages = [m \in {} |-> 0]
        /\ InitServerVars
        /\ InitCandidateVars
        /\ InitLeaderVars
        /\ InitLogVars
        /\ InitAuxVars

----
\* Define state transitions
Restart(i) ==
    /\ restartCtr < MaxRestarts
    /\ state'           = [state EXCEPT ![i] = Follower]
    /\ votesGranted'    = [votesGranted EXCEPT ![i] = {}]
    /\ nextIndex'       = [nextIndex EXCEPT ![i] = [j \in Server |-> 1]]
    /\ matchIndex'      = [matchIndex EXCEPT ![i] = [j \in Server |-> 0]]
    /\ pendingResponse' = [pendingResponse EXCEPT ![i] = [j \in Server |-> FALSE]]
    /\ commitIndex'     = [commitIndex EXCEPT ![i] = 0]
    /\ restartCtr'      = restartCtr + 1
```

```
    /\ UNCHANGED <<messages, currentTerm, votedFor, log, acked, electionCtr>>

RequestVote(i) ==
    /\ electionCtr < MaxElections
    /\ state[i] \in {Follower, Candidate}
    /\ state' = [state EXCEPT ![i] = Candidate]
    /\ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
    /\ votedFor' = [votedFor EXCEPT ![i] = i] \* votes for itself
    /\ votesGranted'   = [votesGranted EXCEPT ![i] = {i}] \* votes for itself
    /\ electionCtr' = electionCtr + 1
    /\ SendMultipleOnce(
           {[mtype         |-> RequestVoteRequest,
             mterm         |-> currentTerm[i] + 1,
             mlastLogTerm  |-> LastTerm(log[i]),
             mlastLogIndex |-> Len(log[i]),
             msource       |-> i,
             mdest         |-> j] : j \in Server \ {i}})
    /\ UNCHANGED <<acked, leaderVars, logVars, restartCtr>>

AppendEntries(i, j) ==
    /\ i /= j
    /\ state[i] = Leader
    /\ pendingResponse[i][j] = FALSE \* not already waiting for a response
    /\ LET prevLogIndex == nextIndex[i][j] - 1
           prevLogTerm == IF prevLogIndex > 0 THEN
                              log[i][prevLogIndex].term
                          ELSE
                              0
           \* Send up to 1 entry, constrained by the end of the log.
           lastEntry == Min({Len(log[i]), nextIndex[i][j]})
           entries == SubSeq(log[i], nextIndex[i][j], lastEntry)
       IN
          /\ pendingResponse' = [pendingResponse EXCEPT ![i][j] = TRUE]
          /\ Send([mtype          |-> AppendEntriesRequest,
                   mterm          |-> currentTerm[i],
                   mprevLogIndex  |-> prevLogIndex,
                   mprevLogTerm   |-> prevLogTerm,
                   mentries       |-> entries,
                   mcommitIndex   |-> Min({commitIndex[i], lastEntry}),
                   msource        |-> i,
                   mdest          |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, nextIndex, matchIndex, logVars, auxVars>>

BecomeLeader(i) ==
    /\ state[i] = Candidate
    /\ votesGranted[i] \in Quorum
    /\ state'      = [state EXCEPT ![i] = Leader]
    /\ nextIndex'  = [nextIndex EXCEPT ![i] =
                          [j \in Server |-> Len(log[i]) + 1]]
    /\ matchIndex' = [matchIndex EXCEPT ![i] =
                          [j \in Server |-> 0]]
    /\ pendingResponse' = [pendingResponse EXCEPT ![i] =
                               [j \in Server |-> FALSE]]
    /\ UNCHANGED <<messages, currentTerm, votedFor, candidateVars,
                   auxVars, logVars>>

AdvanceCommitIndex(i) ==
    /\ state[i] = Leader
    /\ LET \* The set of servers that agree up through index.
           Agree(index) == {i} \cup {k \in Server :
                                       /\ matchIndex[i][k] >= index }
           \* The maximum indexes for which a quorum agrees
           agreeIndexes == {index \in 1..Len(log[i]) :
                               Agree(index) \in Quorum}
           \* New value for commitIndex'[i]
           newCommitIndex ==
              IF /\ agreeIndexes /= {}
                 /\ log[i][Max(agreeIndexes)].term = currentTerm[i]
              THEN
```

```
                Max(agreeIndexes)
            ELSE
                commitIndex[i]
       IN
          /\ commitIndex[i] < newCommitIndex \* only enabled if it actually advances
          /\ commitIndex' = [commitIndex EXCEPT ![i] = newCommitIndex]
          /\ acked' = [v \in Value |->
                        IF acked[v] = FALSE
                        THEN v \in { log[i][index].value : index \in commitIndex[i]+1..newCommitIndex }
                        ELSE acked[v]]
    /\ UNCHANGED <<messages, serverVars, candidateVars, leaderVars, log,
                   pendingResponse, electionCtr, restartCtr>>

UpdateTerm ==
    \E m \in DOMAIN messages :
        /\ m.mterm > currentTerm[m.mdest]
        /\ currentTerm'    = [currentTerm EXCEPT ![m.mdest] = m.mterm]
        /\ state'          = [state       EXCEPT ![m.mdest] = Follower]
        /\ votedFor'       = [votedFor    EXCEPT ![m.mdest] = Nil]
           \* messages is unchanged so m can be processed further.
        /\ UNCHANGED <<messages, candidateVars, leaderVars, logVars, auxVars>>

HandleRequestVoteRequest ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, RequestVoteRequest, LessOrEqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == \/ m.mlastLogTerm > LastTerm(log[i])
                        \/ /\ m.mlastLogTerm = LastTerm(log[i])
                           /\ m.mlastLogIndex >= Len(log[i])
               grant == /\ m.mterm = currentTerm[i]
                        /\ logOk
                        /\ votedFor[i] \in {Nil, j}
           IN /\ m.mterm <= currentTerm[i]
              /\ \/ grant  /\ votedFor' = [votedFor EXCEPT ![i] = j]
                 \/ ~grant /\ UNCHANGED votedFor
              /\ Reply([mtype        |-> RequestVoteResponse,
                        mterm        |-> currentTerm[i],
                        mvoteGranted |-> grant,
                        msource      |-> i,
                        mdest        |-> j],
                        m)
              /\ UNCHANGED <<state, currentTerm, candidateVars, leaderVars,
                             logVars, auxVars>>

HandleRequestVoteResponse ==
    \E m \in DOMAIN messages :
        \* This tallies votes even when the current state is not Candidate, but
        \* they won't be looked at, so it doesn't matter.
        /\ ReceivableMessage(m, RequestVoteResponse, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
           IN
              /\ \/ /\ m.mvoteGranted
                    /\ votesGranted' = [votesGranted EXCEPT ![i] =
                                            votesGranted[i] \cup {j}]
                 \/ /\ ~m.mvoteGranted
                    /\ UNCHANGED <<votesGranted>>
              /\ Discard(m)
              /\ UNCHANGED <<serverVars, votedFor, leaderVars, logVars,
                             auxVars>>
LogOk(i, m) ==
    \/ m.mprevLogIndex = 0
    \/ /\ m.mprevLogIndex > 0
       /\ m.mprevLogIndex <= Len(log[i])
       /\ m.mprevLogTerm = log[i][m.mprevLogIndex].term

RejectAppendEntriesRequest ==
    \E m \in DOMAIN messages :
```

```
        /\ ReceivableMessage(m, AppendEntriesRequest, LessOrEqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == LogOk(i, m)
           IN  /\ \/ m.mterm < currentTerm[i]
                  \/ /\ m.mterm = currentTerm[i]
                     /\ state[i] = Follower
                     /\ \lnot logOk
               /\ Reply([mtype           |-> AppendEntriesResponse,
                         mterm           |-> currentTerm[i],
                         msuccess        |-> FALSE,
                         mmatchIndex     |-> 0,
                         msource         |-> i,
                         mdest           |-> j],
                         m)
               /\ UNCHANGED <<state, candidateVars, leaderVars, serverVars,
                              logVars, auxVars>>

CanAppend(m, i) ==
    /\ m.mentries /= << >>
    /\ Len(log[i]) = m.mprevLogIndex

NeedsTruncation(m, i, index) ==
   \/ /\ m.mentries /= <<>>
      /\ Len(log[i]) >= index
   \/ /\ m.mentries = <<>>
      /\ Len(log[i]) > m.mprevLogIndex

TruncateLog(m, i) ==
    [index \in 1..m.mprevLogIndex |-> log[i][index]]

AcceptAppendEntriesRequest ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, AppendEntriesRequest, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
               logOk == LogOk(i, m)
               index == m.mprevLogIndex + 1
           IN
              /\ state[i] \in { Follower, Candidate }
              /\ logOk
              /\ LET new_log == CASE CanAppend(m, i) ->
                                        [log EXCEPT ![i] = Append(log[i], m.mentries[1])]
                                  [] NeedsTruncation(m, i , index) /\ m.mentries # <<>> ->
                                        [log EXCEPT ![i] = Append(TruncateLog(m, i), m.mentries[1])]
                                  [] NeedsTruncation(m, i , index) /\ m.mentries = <<>> ->
                                        [log EXCEPT ![i] = TruncateLog(m, i)]
                                  [] OTHER -> log
                 IN
                    /\ state' = [state EXCEPT ![i] = Follower]
                    /\ commitIndex' = [commitIndex EXCEPT ![i] =
                                              m.mcommitIndex]
                    /\ log' = new_log
                    /\ Reply([mtype           |-> AppendEntriesResponse,
                              mterm           |-> currentTerm[i],
                              msuccess        |-> TRUE,
                              mmatchIndex     |-> m.mprevLogIndex +
                                                  Len(m.mentries),
                              msource         |-> i,
                              mdest           |-> j],
                              m)
                    /\ UNCHANGED <<candidateVars, leaderVars, votedFor, currentTerm,
                                   auxVars>>

HandleAppendEntriesResponse ==
    \E m \in DOMAIN messages :
        /\ ReceivableMessage(m, AppendEntriesResponse, EqualTerm)
        /\ LET i     == m.mdest
               j     == m.msource
```

```
           IN
              /\ \/ /\ m.msuccess \* successful
                    /\ nextIndex'  = [nextIndex  EXCEPT ![i][j] = m.mmatchIndex + 1]
                    /\ matchIndex' = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
                 \/ /\ \lnot m.msuccess \* not successful
                    /\ nextIndex' = [nextIndex EXCEPT ![i][j] =
                                        Max({nextIndex[i][j] - 1, 1})]
                    /\ UNCHANGED <<matchIndex>>
              /\ pendingResponse' = [pendingResponse EXCEPT ![i][j] = FALSE]
              /\ Discard(m)
              /\ UNCHANGED <<serverVars, candidateVars, logVars, auxVars>>

----
\* Defines how the variables may transition.
Next ==
        \/ \E i \in Server : Restart(i)
        \/ \E i \in Server : RequestVote(i)
        \/ \E i \in Server : BecomeLeader(i)
        \/ \E i \in Server : AdvanceCommitIndex(i)
        \/ \E i,j \in Server : AppendEntries(i, j)
        \/ UpdateTerm
        \/ HandleRequestVoteRequest
        \/ HandleRequestVoteResponse
        \/ RejectAppendEntriesRequest
        \/ AcceptAppendEntriesRequest
        \/ HandleAppendEntriesResponse

NoStuttering ==
    WF_vars(Next)

Spec == Init /\ [][Next]_vars

LivenessSpec == Init /\ [][Next]_vars /\ NoStuttering

\* INVARIANTS -------------------------
MinCommitIndex(s1, s2) ==
    IF commitIndex[s1] < commitIndex[s2]
    THEN commitIndex[s1]
    ELSE commitIndex[s2]

NoLogDivergence ==
    \A s1, s2 \in Server :
        IF s1 = s2
        THEN TRUE
        ELSE
            LET lowest_common_ci == MinCommitIndex(s1, s2)
            IN IF lowest_common_ci > 0
               THEN \A index \in 1..lowest_common_ci : log[s1][index] = log[s2][index]
               ELSE TRUE

LeaderHasAllAckedValues ==
    \* for every acknowledged value
    \A v \in Value :
        IF acked[v] = TRUE
        THEN
            \* there does not exist a server that
            ~\E i \in Server :
                \* is a leader
                /\ state[i] = Leader
                \* and which is the newest leader (aka not stale)
                /\ ~\E l \in Server :
                    /\ l # i
                    /\ currentTerm[l] > currentTerm[i]
                \* and that is missing the value
                /\ ~\E index \in DOMAIN log[i] :
                    log[i][index].value = v
        ELSE TRUE

=============================================================================
```

# ПРИЛОЖЕНИЕ Б

## Конфигурация модели для спецификация Raft

Листинг Б.2 – Конфигурация модели для спецификации Raft

```
CONSTANTS
    n1 = n1
    n2 = n2
    n3 = n3
    v1 = v1
    Server = { n1, n2, n3}
    Value = { v1 }
    Follower = Follower
    Candidate = Candidate
    Leader = Leader
    Nil = Nil
    RequestVoteRequest = RequestVoteRequest
    RequestVoteResponse = RequestVoteResponse
    AppendEntriesRequest = AppendEntriesRequest
    AppendEntriesResponse = AppendEntriesResponse
    EqualTerm = EqualTerm
    LessOrEqualTerm = LessOrEqualTerm
    MaxElections = 2
    MaxRestarts = 0

INIT Init
NEXT Next

VIEW view
SYMMETRY symmServers

INVARIANT
LeaderHasAllAckedValues
NoLogDivergence
```