ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет Санкт-Петербургская школа**
**физико-математических и компьютерных наук**

Лупуляк Ольга Сергеевна

# ДЕКЛАРАТИВНОЕ ТЕСТИРОВАНИЕ ВЫСОКОУРОВНЕВЫХ СВОЙСТВ РАСПРЕДЕЛЁННЫХ АЛГОРИТМОВ

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
А. К. Сатарин

Руководитель
д-р физ.-мат. наук
А.В. Омельченко

Консультант
Н.Д. Коваль

Санкт-Петербург 2021

# Оглавление

Так как распределённые системы используются в наши дни повсеместно, тестирование распределённых систем является важной задаче. В этой работе представлен фреймворк, позволяющий эмулировать и тестировать распределённые системы в разных режимах. Фреймворк предоставляет пользователю возможность указывать лишь основные свойства системы, такие как виды возможных отказов, и не требует написания конкретных сценариев тестирования. Несмотря на то, что инструмент не может применяться к реальным распределённым системам, он может быть полезен для изучения распределённых алгоритмов начинающими программистами и для прототипирования промышленных распределённых систем.

**Ключевые слова:** распределённые системы, тестирование, проверка моделей, эмулирование распределённых систем

Nowadays, distributed systems are widely used, so testing distributed algorithms is an important problem. In this work we present a framework which emulates a distributed system and tests multiple execution scenarios using stress and model checking strategies. The framework provides a declarative way of testing distributed systems, by specifying the system properties (for example, network reliability), operations to be examined, number of iterations and so on. Although the framework does not allow the development of real distributed code that can be run on multiple machines, it can be useful for novice programmers studying distributed algorithms and for researchers for testing concepts of newly developed distributed algorithms.

**Keywords:** distributed systems, stress testing, model checking, emulation of distributed systems

# Введение

## Актуальность работы

Распределённая система – это система, состоящая из независимых узлов, выполняющих общую задачу. У узлов нет общей памяти, и они обмениваются сообщениями для синхронизации. Узлы могут быть физически разнесены в пространстве и выполняться на разных компьютерах. Распределённые системы используются в наши дни повсеместно. Они позволяют обрабатывать большие объёмы данных, предоставляют возможность удалённого доступа к данным, обеспечивают большую надёжность за счёт устойчивости к отказам отдельных узлов. Фактически, интернет представляет из себя одну большую распределённую систему.

Распределённые алгоритмы описывают взаимодействие узлов внутри распределённой системы друг с другом и обеспечивают корректную работы системы. Распределённые алгоритмы часто достаточно сложны по своей логике, так как требуется обеспечить устойчивость к отказам узлов и задержкам при доставке сообщений. Согласно исследованию [22] большая часть ошибок возникает из-за ошибок в алгоритме. Поиск подобных ошибок в промышленных распределённых системах усложняется тем, что в таких системах много технически сложного кода, и требуется отделять алгоритмические ошибки от технических. Реализация распределённых алгоритмов в отрыве от распределённых систем, с минимальным количеством технических деталей, и тщательное тестирование полученной реализации помогло бы сократить число алгоритмических ошибок, возникающих в реальных распределённых системах. Также начинающим программистам, изучающим распределённые алгоритмы, полезно реализовывать эти алгоритмы самостоятельно, при этом не тратя много сил на технические сложности, которые нерелевантны изучаемому материалу.

Недостаточно тщательное тестирование делает подобные реализации распределённых алгоритмов бесполезными, так как в таком случае

сложные ошибки в алгоритме не будут найдены. Тестирование алгоритма при этом не может ограничиваться несколькими модульными тестами. Даже если в системе не может возникать никаких отказов, существует большое количество вариантов упорядочивания событий, таких как получение сообщения, начало операции и так далее. Ошибка может проявляться далеко не во всех из них. Если алгоритм был запущен всего несколько раз, существует достаточно высокий шанс того, что в алгоритме присутствуют ошибки, которые не были найдены. Также в случае распределённых алгоритмов под корректностью часто понимают то, что результаты удовлетворяют какой-то модели согласованности (например, линеаризуемости). Модель согласованности для параллельных алгоритмов подразумевает, что у одной операции могут быть разные результаты, которые будут являться корректными. Поэтому проверка того, удовлетворяют ли результаты операций модели согласованности, является отдельной задачей. Кроме того, если алгоритм должен корректно обрабатывать какие-то отказы узлов или сети, то требуется чтобы при тестировании эти отказы происходили. При этом не должно происходить отказов, для которых алгоритм не даёт гарантий корректного исполнения.

Обычно в таких случаях самостоятельно реализуют наивную тестирующую систему, заточенную под конкретный тестируемый алгоритм, которая требует описывать в самом тесте, что должно происходить с системой (например, в какие моменты исполнения какие узлы должны отказать). Однако такой подход требует значительных трудозатрат, так как помимо реализации самой тестирующей системы требуется самостоятельно придумывать и реализовывать различные тестовые сценарии. Наличие инструмента, не требующего составления тестовых сценариев, позволило бы значительно упростить тестирование. Дополнительным преимуществом была бы возможность минимизировать сценарий, на котором воспроизводится ошибка, чтобы упростить отладку. Подобный инструмент, Lincheck [19], существует для многопоточных алгоритмов. Это декларативный инструмент для тестирования многопоточных структур данных под JVM. Он отличается тем, что не требует

от пользователя написания конкретных сценариев тестирования. Вместо этого пользователю требуется указать высокоуровневую информацию, например выделить при помощи специальных аннотаций методы, которые требуется протестировать. После этого происходит автоматическая генерация многопоточного сценария, который затем исполняется в stress и model checking режиме, в процессе исполнения собираются результаты исполнения, которые затем проверяются на линеаризуемость. Распределённые и многопоточные алгоритмы относятся к классу параллельных алгоритмов, что делает возможным переиспользование части функциональности, поэтому этот фреймворк был взят для основы разрабатываемого инструмента для декларативного тестирования распределённых алгоритмов.

## Цель и задачи

Целью данной работы является разработка декларативного инструмента, позволяющего тестировать распределённые алгоритмы, на основе Lincheck. Для этого ставятся следующие задачи:

- Разработка программного интерфейса для реализации распределённых алгоритмов и для конфигурации параметров распределённых систем
- Реализация stress режима для тестирования распределённых алгоритмов
- Адаптация существующего в Lincheck режима model checking под многопоточные алгоритмы
- Тестирование полученного инструмента при помощи реализации различных распределённых алгоритмов, как корректных, так и содержащих ошибки

## Достигнутые результаты

В рамках данной работы была расширена функциональность Lincheck возможностью декларативного тестирования распределённых алгоритмов. Для этого было реализовано API, позволяющее тестировать распределённые алгоритмы при помощи Lincheck, а также поддержаны как stress режим, так и model checking. Полученный инструмент был протестирован на множестве существующих распределённых алгоритмов, реализованных на основе предоставляемого API, и в которые намеренно были внедрены ошибки.

## Структура работы

В главе 1 представлен обзор существующих инструментов для тестирования распределённых систем. В главе 2 перечислены требования, которым должен соответствовать реализуемый инструмент. В главе 3 описан программный интерфейс, который был разработан для реализации и тестирования распределённого алгоритма. В главе 4 рассказано про реализацию stress режима тестирования. В главе 5 описана реализация режима тестирования model checking. В главе 6 демонстрируется работоспособность полученного инструмента, применимость его для тестирования различных алгоритмов, а также эффективность в нахождении ошибок. В заключении представлены анализ проделанной работы и возможные перспективы дальнейшей деятельности.

# 1. Обзор литературы

## 1.1. Существующие инструменты для тестирования распределённых алгоритмов

Существует множество инструментов, проверяющих корректность распределённых алгоритмов. В этой работе будут упомянуты лишь самые известные, а также те, которые близки по тематике к данной работе.

Формальные методы и верификаторы эффективно применяются для доказательства корректности распределённых систем [13, 25]. TLA+[6] использовался некоторыми компаниями для верификации их распределённых систем [20]. Недостатки TLA+ и других подобных специальных языков заключаются в том, что они применяются к моделями, которые слишком сильно отличаются от реальной реализации распределённых систем. Помимо этого, эти инструменты требуют высокого порога вхождения и не подходят для обучения распределённым алгоритмам.

Самым широко известным инструментом для тестирования промышленных распределённых систем является Jepsen [5]. Пользователю требуется реализовать код клиента на языке программирования Clojure, а также настроить другие параметры тестирования, такие как генерация отказов и проверка результатов. По мере исполнения в систему добавляются различные отказы узлов и сети. Операции, выполняемые распределённой системой, генерируются автоматически, а результаты их проверяются на соответствие заданной модели согласованности. Также существует инструмент Maelstorm [11], сделанный на основе Jepsen и эмулирующий распределённую систему. Он используется в качестве учебного пособия к Jepsen и позволяет тестировать лишь фиксированный набор алгоритмов.

Несмотря на то, что Jepsen предоставляет широкий круг возможностей, а также доказал свою эффективность в поиске ошибок, однако для его использования требуется много предварительных знаний, в

частности владение языком Clojure и достаточно низкоуровневым API, предоставляемым Jepsen.

В работе [15] представлен подход PCTCP (Probabilistic Concurrency Testing with Chain Partitioning) для распределённых алгоритмов. В нём события, происходящие в системе, разбиваются на цепочки. Внутри каждой цепочки события имеют строгий линейный порядок. У каждой цепочки есть свой приоритет, который меняется по ходу исполнения. Операции выполняются из цепочки, у которой стоит наименьший приоритет.

Инструмент Morpheous [23] тестирует распределённые системы, реализованные на языке программирования Erlang, используя технику partial order sampling [24]. Несмотря на то, что инструмент удобен тем, что Erlang предоставляет встроенную в язык поддержку операций, связанных с распределёнными системами, Morpheous не поддерживает генерацию сценариев и добавление отказов. Помимо этого, инструмент не применим для обучения студентов, так как Erlang относится к функциональным языкам программирования, что может вызвать трудности у начинающих программистов.

Популярным подходом к тестированию распределённых систем также является model checking. Model checking означает перебор всех возможных вариантов исполнения с какими-то ограничениями. Такие техники, как dynamic partial-order reduction [2] и dynamic interface reduction [14] уменьшают количество вариантов, которые следует перебрать, используя то, что многие операции независимы в параллельных алгоритмах.

К наиболее известным инструментам для model checking относятся MODIST [10], MaceMC [9] и SAMC [17]. MODIST тестирует немодифицированные распределённые системы, добавляя различные помехи в сеть и отказы узлов. MaceMC исследует свойства живучести, используя ограниченный поиск в глубину и случайные блуждания. SAMC использует информацию о семантике сообщений, чтобы уменьшить объём исследуемых вариантов. Однако все эти инструменты разрабатывались достаточно давно, найти их в открытом доступе не удалось и, по всей

видимости, на текущий момент они не поддерживаются. FlyMC [3] – это ещё один model checker, он работает эффективнее по сравнению с упомянутыми ранее инструментами для model checkinhg, однако этот инструмент достаточно сложно настраивать, и он не обладает достаточно удобным интерфейсом.

Ещё один инструмент для model checking – это DSLabs [18]. Он был разработан для тестирования домашних заданий студентов на курсе распределённых систем, поэтому он легко устанавливается и имеет удобный программный интерфейс для реализации распределённых алгоритмов. Он проверяет выполнение предикатов для состояния всей системы и находит наименьший след, для которого предикат нарушается. Также в процессе исследований состояний системы происходит отсечение некоторых состояний, которые можно не исследовать. Эта техника требует от пользователя, реализующего набор тестов, определение и написание предикатов для состояний системы, которые позволили бы применить эти техники, и тем самым приводит к тому, что тесты занимают слишком много строк кода. К примеру, тесты для алгоритма Paxos [8] занимают более 1000 строк кода [1].

## 1.2. Lincheck

Как было отмечено выше, Lincheck – это декларативный фреймворк для тестирования многопоточных алгоритмов. По заданным пользователем параметрам Lincheck автоматически создаёт конфигурацию теста, генерирует сценарии и запускает алгоритм. После исполнения результаты проверяются на линеаризуемость, в качестве последовательного исполнения берётся сама тестируемая структура или указанная пользователем последовательная спецификация. Помимо этого пользователь может задавать произвольные функции для валидации, которые будут вызваны после исполнения. Фреймворк сообщает об ошибке в случае непредвиденного исключения, зависания теста, неверных результатов выполнения или исключения, выпавшего во время выполне-

[1] Тест для алгоритма Paxos на основе DSLabs: https://github.com/emichael/dslabs/blob/master/labs/lab3-paxos/tst/dslabs/paxos/PaxosTest.java

ния функций для валидации. В случае, если ошибка найдена, фреймворк пытается найти минимальный сценарий, при котором эта ошибка воспроизводится, чтобы упростить отладку для пользователя.

В Lincheck существует два режима для тестирования. Первый – это stress режим, в котором много раз запускается один сценарий для нескольких потоков. За счёт случайного переключения потоков можно найти исполнение, которое приводит к ошибке. Второй режим – это bounded model checking. В этом режиме строится дерево различных исполнений и перебираются все возможные варианты исполнения с ограничением на число переключений между потоками.

## 1.3. Выводы и результаты главы

Были рассмотрены различные инструменты для тестирования параллельных алгоритмов. Среди инструментов для тестирования распределённых алгоритмов близким по функционалу к желаемому инструменту является Maelstorm. Однако необходимость знания языка Clojure значительно усложняет его использование. Помимо этого, в этом инструменте нет возможности минимизировать ошибку, то есть найти вариант исполнения с наименьшим числом действий, при котором ошибка воспроизводится.

Lincheck является удобным многофункциональным инструментом, позволяющим тестировать многопоточные алгоритмы при помощи очень простого интерфейса, который может быть освоен пользователем любого уровня. Он тестирует структуры, реализованные на JVM языках программирования, таких как Java или Kotlin. Это императивные языки, что делает их более доступными для обычного пользователя по сравнению с функциональными языками, такими как Clojure или Erlang.

# 2. Требования к инструменту

Для начала требовалось разработать API, которое пользователь мог бы использовать для реализации и тестирования распределённых алгоритмов. API для реализации требуется для того, чтобы позволить пользователю писать как можно более высокоуровневые реализации, не заботясь о работе с сетью и прочих технических деталях. При этом этот программный интерфейс должен быть интуитивно понятным. API для написания тестов должно давать возможность пользователю указывать какие-то общие свойства системы, и не требовать от него конкретных сценариев. При этом задачи тестировать реальные распределённые системы не стояло, поэтому фреймворк не может использоваться для реализации настоящих распределённых систем.

## 2.1. Узлы, участвующие в системе

Исходно в Lincheck тестировалась некоторая потокобезопасная структура, для которой пользователь должен был указать тестируемые методы при помощи специальных аннотаций. В случае распределённых алгоритмов логично было бы сделать так, чтобы пользователь определял класс, соответствующий типу узла, и во время тестирования создавалось бы несколько экземпляров этого класса. При этом в отличие от многопоточных структур, для которых создавался Lincheck, в распределённом алгоритме могут участвовать узлы, выполняющие разные функции (например, клиенты и сервера).

## 2.2. Отсутствие многопоточности для одного узла

В настоящих распределённых системах на одном узле могут работать несколько потоков. Однако это требует думать о потокобезопасности, что добавляет дополнительные технические сложности, которые не имеют отношение к логике алгоритма. Чтобы этого избежать, требуется, чтобы все действия, происходящие на узле, выполнялись в одном потоке.

## 2.3. Операции

Одной из важных проблем было определение того, как именно устроены операции. Операции в распределённых алгоритмов чаще всего выглядят следующим образом: узел отправляет какие-то запросы другим узлам, далее ждёт ответов от других узлов, которые позволили бы ему завершить операцию и вернуть результат. Существовало несколько вариантов того, как можно было бы определить операции:

1. Есть отдельный вид сообщений, определяющий начало операции. После того, как узел получил такое сообщение, он начал выполнение этой операции. Когда операция заканчивается, результат отправляется как ответ на это сообщение.

2. Определять операции как методы класса, обозначающего узел, так же, как это было сделано в Lincheck для многопоточных алгоритмов. В таком случае операция должна была приостанавливаться, ожидая результата, а потом продолжаться после того, как это становится возможно.

Решено было остановиться на втором способе, так как он позволяет использовать генерацию сценария и проверку результатов на основе существования последовательной истории. В первом случае требовался отдельный интерфейс для того, чтобы установить понятие последовательной истории для сообщений и их ответов. Помимо этого, второй способ задания операций был более гибким: можно отдельно реализовать класс, соответствующий клиенту, и определять операции в нём (как обычно и происходит в распределённых системах), а можно было использовать только один тип узлов, и определять операции прямо для него, если не хочется отдельно реализовывать класс, который будет заниматься только тем, что отправлять запросы и получать ответы.

## 2.4. Адреса узлов и топология сети

В распределённых алгоритмах узлы взаимодействуют друг с другом при помощи сообщений. Содержание сообщений может быть разным в

зависимости от алгоритма. Для того, чтобы иметь возможность отправлять сообщения, для каждого узла должен существовать уникальный идентификатор. Было решено в качестве идентификатора использовать числа от 0 до $n-1$, где $n$ – это число узлов в системе. Изначально каждый узел получает информацию о своём номере и об общем количестве узлов, а также возможность узнать адреса узлов конкретного типа. Такое решение накладывает следующие ограничения:

1. Не поддерживаются алгоритмы, где требуется распространять информацию между узлами об адресах других узлов.

2. Не поддерживается динамическое добавление узлов в систему.

3. Не поддерживается сложная топология сети. Любой узел может написать любому другому узлу.

## 2.5. Периодические таймеры

Некоторые алгоритмы предполагают существование периодического процесса для узла, который происходит независимо от отправляемых и получаемых сообщений. Например, в алгоритме распределённого консенсуса Raft [12] узел, который на данный момент является лидером, периодически рассылает heartbeat сообщения другим узлам. Чтобы не нарушать условие о том, что все действия узла должны выполняться в одном потоке, для установки и удаления таймеров требовалось предоставить специальное API.

## 2.6. История событий

Для упрощения отладки в случае нахождения ошибки требуется сохранять историю всех событий, происходивших в системе. Под событием понимается отправка сообщения, получение сообщения, начало выполнения операции, отказ узла, восстановление узла, установка, срабатывание и удаление таймера, а также произвольное внутреннее

событие, которое пользователь может зафиксировать в какой-то точке исполнения программы. Такие внутренние события могут помогать в отладке, играя роль отладочного вывода, а также позволяют проверять произвольные инварианты, связанные с исполнением. Например, в случае поиска минимального согласованного среза, для которого выполняется некоторый предикат, таким событием может служить то, что предикат выполнился для какого-то узла.

К каждому событию прикладываются соответствующие этому событию векторные часы, а также состояние узла, к которому относится событие. Состояние узла возвращается функцией, переопределяемой пользователем. В ней могут быть текущие значения изменяемых переменных внутри узла.

## 2.7. Определение корректности

Для распределённых алгоритмов используют разные модели согласованности [21]. В этой работе было решено проверять линеаризуемость результатов операций, так как проверка на линеаризуемость уже есть в Lincheck. Другие модели согласованности не реализованы из-за нехватки времени.

Также после исполнения проверяются произвольные инварианты, определённые пользователем. Для этого он реализует функции, которые будут вызваны после исполнения, и которые должны бросать исключение в случае, если какой-то вариант не выполнился. Такая функциональность уже тоже была в Lincheck. Однако, в случае распределённых алгоритмов может быть важно иметь доступ к истории событий, происходивших в процессе исполнения (таких как отправка и получения сообщения), а также иметь возможность проверить свойство, связанное со всеми узлами (например, что какой-то предикат имеет одинаковое значение для всех узлов). При помощи этих функций, а также имея возможность сохранять в истории внутренние события, пользователь получает возможность реализовать самостоятельно проверку произвольных инвариантов.

## 2.8. Помехи в сети

При пересылке сообщений в сети могут возникать всевозможные помехи, такие как потеря сообщений, дублирование сообщений или отсутствие FIFO порядка. Отдельно идут так называемые разделения сети (network partitions), когда связь между двумя узлами теряется на какое-то продолжительное время. Распределённые алгоритмы разрабатываются для разных моделей распределённых систем, некоторые из которых предполагают какие-то типы отказов, происходящих в сети. Поэтому требуется дать возможность пользователю указать, какие виды отказов бывают в системе, и искусственно проэмулировать эти отказы.

## 2.9. Отказы узлов

В процессе исполнения могут происходить отказы. Некоторые алгоритмы предполагают, что после отказа узел не может восстановиться и продолжить работать. Другие поддерживают восстановление узлов после отказа. В таких случаях у узла должно быть персистентное хранилище, позволяющее частично восстановить своё состояние. Значения, хранящиеся в персистентном хранилище, могут быть произвольными в зависимости от алгоритма. Также по аналогии с базами данных должно быть удобно делать сложные запросы к этому персистентному хранилищу.

## 2.10. Выводы и результаты главы

Были сформулированы основные требования, которым должен соответствовать разрабатываемый инструмент, чтобы с его помощью можно было реализовывать и тестировать различные распределённые алгоритмы. В отличие от многопоточных алгоритмов, поддерживаемых Lincheck, в распределённом алгоритме может участвовать несколько узлов, некоторые из которых могут быть разных типов. Инструмент должен эмулировать работу с сетью, то есть поддерживать получение

и отправку сообщений и возможность узнавать адреса других узлов. Пользователь может указывать, какие отказы могут случаться в системе: потеря, дублирование и переупорядочивание сообщений, разделение сети, а также отказы и восстановления узлов. Чтобы избавить пользователя от возможных ошибок, связанных с многопоточностью, гарантируется, что все действия, происходящие на одном узле, выполняются в одном потоке.

# 3. Программный интерфейс

Требовалось предоставить программный интерфейс для реализации распределённых алгоритмов и для реализации тестов.

## 3.1. Асинхронные операции и корутины

В секции 2.3 упоминалось, что операции в распределённых системах чаще всего асинхронны. Требовался механизм, который позволил бы приостановить исполнение операции до тех пор, пока не был получен ответ, и после этого вернуть результат. В процессе ожидания результата узел должен обрабатывать приходящие сообщения. При этом обработка сообщений должна происходить в том же потоке, что и выполнение операций, чтобы не нарушать требование из секции 2.2. С учётом этих ограничений оказалось удобно использовать корутины.

Корутины – это легковесные потоки, позволяющие выполнять асинхронные задачи, которые есть в Kotlin[2]. Корутины не привязаны к нативным потокам, в одном нативном потоке могут выполняться сразу несколько корутин. Корутины могут приостанавливаться в специальных точках, а потом продолжить исполнение после того, как будут выполнены условия, позволяющие исполнению продолжиться. Функции, содержащие эти точки остановки, помечены специальным ключевым словом `suspend`. Такие функции могут быть вызваны только внутри корутин.

Listing 1: Пример `suspend` функции

```
1 suspend fun foo() : Int {
2     // suspends until the data is sent to the channel
3     val i = channel.receive()
4     return i
5 }
```

В примере 1 функция `foo()` получает число при помощи канала и возвращает его. Каналы[3] – это специальный инструмент в Kotlin, позволяющий безопасно обмениваться данными между корутинами. Исполне-

[2]Корутины в Kotlin: https://github.com/Kotlin/kotlinx.coroutines

[3]Каналы в Kotlin: https://kotlinlang.org/docs/channels.html

ние приостанавливается на методе `receive()` до тех пор, пока в канале не появятся данные, которые можно получить.

Использование корутин предполагает, что алгоритмы будут реализованы именно на языке Kotlin, а не на других JVM языках.

## 3.2. Программный интерфейс для реализации алгоритмов

Для реализации алгоритмов использовался достаточно стандартный подход: узел соответствует экземпляру класса, реализованного пользователем. Для специфичных действий, таких как получение адресов узлов и отправка сообщений, в параметр конструктора передаётся специальная сущность `Environment` (listing 2). Это интерфейс, который реализован по-разному для разных режимов тестирования (stress и model checking).

Listing 2: Интерфейс Environment

```
interface Environment<Message,  Log> {
    // Node adressess
    val nodeId : Int
    val numberOfNodes: Int
    fun getAddressesForClass(cls: Class<out Node<Message>>): List<Int>

    // Send messages
    fun send(msg: Message, receiver: Int)
    fun broadcast(msg: Message, sendToItself: Boolean = false)

    // Persistent storage
    val persistentStorage : MutableList<Log>

    // Record any event
    fun logEvent(attachement: Any?)

    // Periodic timers
    fun setTimer(name: String, ticks: Int, f : () -> Unit)
    fun cancelTimer(name: String)

    // Awaits with timeout
    suspend fun withTimeout(ticks: Int, f: suspend () -> Unit)
}
```

`Environment` параметризируется классом, представляющим собой тип сообщений, которые отправляются между узлами, и классом, представляющим собой запись в персистентное хранилище. В случае, когда со-

общения бывают разных типов, удобно использовать `sealed` классы[4], то есть классы, у которых может быть фиксированное количество наследников и для которых удобно делать pattern matching.

С помощью `Environment` узел может узнать свой собственный порядковый номер, общее число узлов, которые есть в системе, а также порядковые номера узлов, представленных конкретным классом (что может быть нужно, если в системе есть несколько типов узлов, реализованных при помощи разных классов).

Сообщения отправляются при помощи метода `send()`. В параметры передаётся сообщение, которое нужно отправить, и номер узла-получателя. Для случаев, когда требуется разослать сообщение всем узлам, предоставляется метод `broadcast()` (флаг `sendToItself` говорит нужно ли отправлять сообщение себе).

Персистентное хранилище эмулируется при помощи изменяемого списка. Это позволяет использовать стримы для списков, которые есть в Kotlin (такие как `map`, `filter` и так далее), и тем самым позволяет легко и удобно получать нужные данные из персистентного хранилища, что делает его схожим с базой данных, к которой можно делать запросы при помощи SQL (Structured Programming Language). При этом не надо физически сохранять эти данные, что позволяет ускорить исполнение.

Метод `logEvent()` позволяет добавить в произвольный момент исполнения программы запись, которая будет сохранена в истории.

`Environment` предоставляет возможность установки периодических таймеров. У таймера есть уникальное имя, промежуток времени, через который он будет срабатывать (выраженный в абстрактных тиках), и функцию, которую нужно выполнять на каждое срабатывание таймера. Можно в любой момент убрать существующий таймер, указав его имя.

`withTimeout` позволяет форсированно продолжить исполнение, если переданная `suspend` функция приостановилась и не продолжила своё исполнение в указанный промежуток времени. Эту функцию можно использовать для того, чтобы переотправить сообщение в случае, если не был получен ответ.

[4]sealed классы в Kotlin: https://kotlinlang.org/docs/sealed-classes.html

Помимо отправки сообщений требовалось добавить механизм для получения сообщений. Для этого создан специальный интерфейс `Node` (listing 3) с методом `onMessage()`, который вызывается, когда узел получает сообщения. В качестве аргументов передаются сообщение и номер узла-отправителя. Пользователь сам определяет, как следует обрабатывать это сообщение. Помимо этого, в интерфейсе `Node` есть методы, которые пользователь может переопределить по желанию: метод `onStart()` вызывается при запуске узла, метод `onScenarioFinish()` – после выполнения всех операций для данного узла, `recover()` – при восстановлении узла после отказа, а метод `stateRepresentation()` возвращает состояние текущего узла (например, значения каких-то изменяемых полей), которые прикладываются к каждому событию, связанному с этим узлом, и могут помогать при отладке.

Listing 3: Интерфейс Node

```
interface Node<Message> {
    fun onMessage(msg: Message, sender: Int)

    fun recover() {}

    fun onStart() {}

    fun onScenarioFinish() {}

    fun stateRepresentation(): String  = ""
}
```

## 3.3. Параметры тестирования

Требовалось выделить параметры тестирования, которые пользователь может задавать для своей распределённой системы. Эти параметры не должны быть максимально общими, и должны описывать свойства системы, а не свойства конкретного теста. В таблице 1 приведены основные параметры, которые может задавать пользователь для распределённой системы.

Таблица 1: Параметры конфигурации для распределённых алгоритмов

| **Параметр** | **Возможные значения** | **Пояснение** |
|---|---|---|
| loseMessages | `true / false`<br>*default*: `false` | Иногда происходит потеря сообщений |
| duplicate Messages | `true / false`<br>*default*: `false` | Иногда происходит дублирование сообщений |
| messageOrder | `FIFO`,<br>`NO_FIFO`<br><br>*default*: `FIFO` | `FIFO` – сообщения от узла $A$ к узлу $B$ доходят в том же порядке, в каком они были отправлены<br>`NO_FIFO` – сообщения могут переупорядочиваться |
| unavailavle NodesLimit | `(): Int -> Int`<br>*default*: `0` | Максимальное число узлов, которые могут стать недоступны одновременно. Задаётся как функция, принимающая на вход общее число узлов.<br>Можно дополнительно указать класс узла, тогда будет условие на максимальное число отказавших узлов этого класса |

| | | |
|---|---|---|
| networkPartitions | `NONE`,<br>`HALVES`,<br>`SINGLE`<br><br>*default*: `NONE` | Отказы сети (то есть сообщения перестают доходить между узлами).<br>`NONE` – отказов сети не происходит,<br>`HALVES` – сеть делится на компоненты, внутри которых существует связь между любыми двумя узлами<br>`SINGLE` – может теряться связь между произвольными узлами |
| crashMode | `NO_CRASHES`,<br>`NO_RECOVERIES`,<br>`RECOVERIES`,<br>`MIXED`<br><br>*default*:<br>`NO_RECOVERIES` | Режимы отказов узлов.<br>`NO_CRASHES` – отказов узлов не происходит, могут происходить лишь отказы сети<br>`NO_RECOVERIES` – узлы не восстанавливаются после отказа<br>`RECOVERIES` – узлы всегда восстанавливаются после отказа<br>`MIXED` – узлы могут как восстанавливаться после отказа, так и не восстанавливаться |

| | | |
|---|---|---|
| nodeType | `Class<out Node>` | Тип узла, участвующего в системе. Параметр – класс, который представляет этот тип узлов. Можно дополнительно указать максимальное количество узлов такого типа (по умолчанию 3) и минимальное количество узлов этого типа (по умолчанию 1) |

Помимо этих параметров есть ещё общие с многопоточными алгоритмами параметры, такие как число итераций или последовательная спецификация.

## 3.4. Пример

Рассмотрим пример реализации простого key-value хранилища (listing 4), в котором клиенты делают запрос к серверу, чтобы добавить значение с соответствующим ключом и узнать значение по ключу.

Listing 4: Пример реализации распределённого алгоритма при помощи Lincheck

```
1  sealed class Message
2  data class PutRequest(val key: Int, val value: Int) : Message()
3  data class PutResponse(val prevValue: Int?) : Message()
4  data class GetRequest(val key: Int) : Message()
5  data class GetResponse(val value: Int?) : Message()
6
7  class Client(val env: Environment<Message, Unit>) : Node<Message> {
8      private val server = env.getAddressesForClass(Server::class.java)!![0]
9      private val resultsChannel = Channel<Int?>(UNLIMITED)
10
11     override fun onMessage(message: Message, sender: Int) {
12         when (message) {
13             is PutResponse -> resultsChannel.offer(message.prevValue)
14             is GetResponse -> resultsChannel.offer(message.value)
```

```
15          }
16      }
17
18      @Operation
19      suspend fun get(key: Int): Int? {
20          env.send(GetRequest(key), server)
21          return resultsChannel.receive()
22      }
23
24      @Operation
25      suspend fun put(key: Int, value: Int): Int? {
26          env.send(PutRequest(key, value), server)
27          return resultsChannel.receive()
28      }
29  }
30
31  class Server(val env: Environment<Message, Unit>) : Node<Message> {
32      private val storage = mutableMapOf<Int, Int>()
33
34      override fun onMessage(message: Message, sender: Int) {
35          when (message) {
36              is PutRequest -> env.send(PutResponse(storage.put(message.key, message.value)),
                    sender)
37              is GetRequest -> env.send(GetResponse(storage[message.key]), sender)
38          }
39      }
40  }
```

В этом алгоритме фигурируют четыре вида сообщений:

1. `GetRequest` – запрос от клиента серверу для получения значения для ключа `key`.

2. `GetResponse` – возвращает результат на запрос `GetResponse` со значением, соответствующим переданному ключу, или `null`, если такого ключа нет в хранилище.

3. `PutRequest` – запрос от клиента для добавления в хранилище ключа `key` со значением `value`.

4. `PutResponse` – ответ на запрос `PutResponse` с предыдущим значением, которое хранилось под переданным ключом, или `null`, если такого значения не было.

Клиенты узнают адрес сервера при помощи `Environment`. Канал `resultsChannel` используется для того, чтобы передавать из `onMessage` полученный результат (для обеих операций используется один канал, так как операции на одном узле выполняются строго последовательно,

а тип возвращаемого значения в обеих операциях одинаковый). Внутри операций клиенты отправляют запрос серверу, а после этого ждут, пока не будет получен результат. Когда клиенту приходит сообщение, он проверяет, что оно пришло от сервера, и отправляет результат сообщений по каналу, чтобы операция, которая сейчас приостановлена, продолжилась и вернула результат.

Сервер содержит в себе поле типа `MutableMap`[5], в которое он добавляет новые значения и из которого он узнаёт значения для переданного ему ключа.

Простейший тест для этого алгоритма выглядит так (listing 5).

Listing 5: Пример теста к алгоритму с листинга 4

```
class SeqSpec {
    val storage = mutableMapOf<Int, Int>()
    suspend fun put(key : Int, value: Int) = storage.put(key, value)
    suspend fun get(key: Int) = storage[key]
}

class Test {
    @Test
    fun test() = DistributedOptions()
        .sequentialSpecification(SeqSpec::class.java)
        .invocationsPerIteration(3_000)
        .iterations(10)
        .nodeType(Server::class.java, maxNumberOfInstances=1)
        .nodeType(Client::class.java)
        .check()
}
```

Пользователь указывает последовательную спецификацию (она представляет собой просто обёртку над `MutableMap`), количество запусков для одной итерации, количество итераций (то есть сколько сценариев было сгенерировано), а также типы узлов, участвующих в тесте.

Если бы пользователь решил дополнительно протестировать алгоритм на возможность отказов клиентов с последующим восстановлением, тест выглядел бы следующим образом (listing 6).

Listing 6: Пример теста с отказами к алгоритму с листинга 4

```
fun test() = DistributedOptions()
    .sequentialSpecification(SeqSpec::class.java)
    .invocationsPerIteration(3_000)
    .iterations(10)
```

[5]MutableMap: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/

```
5     .nodeType(Server::class.java, maxNumberOfInstances=1)
6     .nodeType(Client::class.java)
7     .unavailableNodesLimit(Client::class.java) { it }
8     .crashMode(RECOVERIES)
9     .check()
```

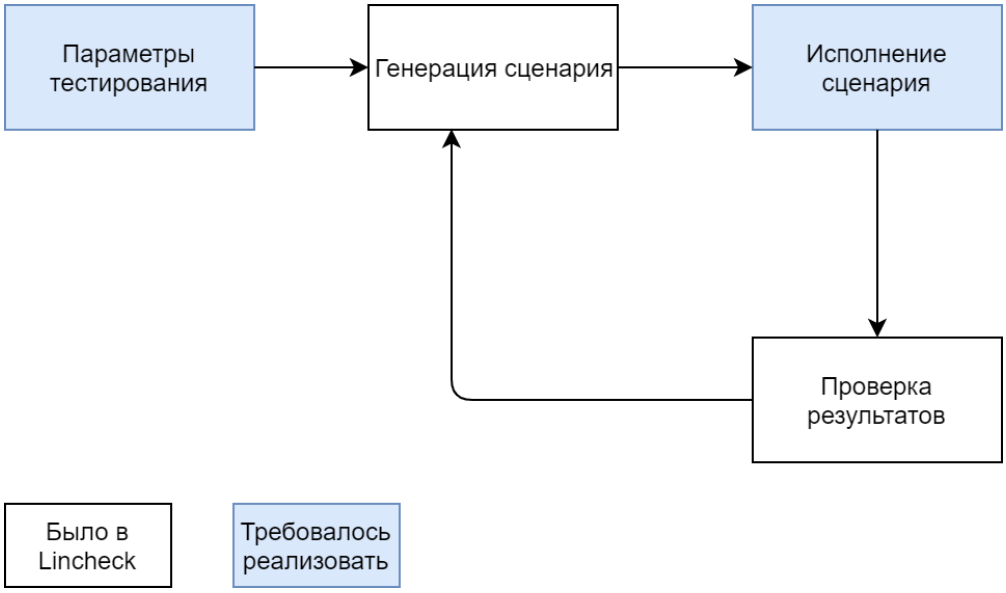
## 3.5. Выводы и результаты главы

Был разработан программный интерфейс для реализации и тестирования распределённых алгоритмов с помощью Lincheck. Для того, чтобы операции можно было выполнить асинхронно, используются корутины из языка Kotlin. Узлы в системе представлены экземплярами классов, реализуемых пользователем. Отправка сообщений, установка периодических таймеров, добавление записей в персистентное хранилище осуществляется при помощи класса `Environment`, передаваемого как параметр конструктора. Класс, соответствующий узлу, наследуется от интерфейса `Node`, чтобы определить, как узел будет обрабатывать сообщения. При конфигурации теста пользователь может указывать типы узлов, участвующие в системе, различные отказы, которые могут случаться в системе, а также максимальное число узлов, которые могут стать одновременно недоступны. Был приведён пример реализации алгоритма при помощи разработанного API, а также примера теста к нему.

# 4. Stress режим тестирования

В Lincheck схема взаимодействия выглядит следующим образом (рисунок 1).

Рис. 1: Взаимодействие компонет в Lincheck




При этом исполнение сценария определяется по-разному для разных режимов тестирования (stress или model checking). Для поддержки распределённых алгоритмов требовалось реализовать заново исполнение сценария.

Было решено не создавать отдельные процессы для запуска узлов, чтобы меньше времени тратить на передачу сообщений, так как в любом случае фреймворк не предназначен для тестирования настоящих распределённых систем. Вместо этого в stress режиме тестирования каждый узел запускается в своём потоке. Это обязывает пользователей не использовать изменяемые статические поля и глобальные переменные в своих реализациях. Также это не позволяет создать слишком много узлов в системе, однако это допустимо, так как нет цели тестировать масштабируемость или производительность.

В Lincheck за исполнение сценария отвечает абстрактный класс `Runner`. От него наследуется `DistributedStressRunner`, который отвечает за исполнение распределённых алгоритмов в stress режиме. Для каждого

сценария создаётся свой экземпляр класса `DistributedStressRunner`, который делает много запусков этого сценария.

## 4.1. Использование корутин в stress режиме

Реализация stress режима активно использует корутины. Для каждого узла запускаются отдельные корутины для выполнения сценария и для получения сообщений (подробнее это описано в секции 4.2).

Диспатчер – это сущность, отвечающая за исполнение корутин. Как было сказано в секции 3.1, корутины могут приостанавливать своё исполнение в специальных точках, называемых точками остановки. Диспатчер принимает на вход блок от одной точки остановки до другой в виде `Runnable` (в дальнейшем будем называть задачей), и дополнительно принимает специальный контекст `CoroutineContext`, содержащий дополнительную информацию о корутине. Для того, чтобы иметь возможность контролировать исполнение в случае отказов и определять момент завершения исполнения (подробнее об этом в секциях 4.4 и 4.8), требовалось сделать свою реализацию диспатчера. Эта реализация содержит внутри себя `Executor` из Java [6], чтобы выполнять задачи. `Executor`-ы создаются один раз для сценария вместе с `Runner` и переиспользуются между запусками. Диспатчеры создаются заново для каждого запуска.

## 4.2. Запуск исполнения

`DistributedStressRunner` на вход получает сценарий и конфигурацию теста. Для каждого узла создаётся экземпляр соответствующего ему класса, а также экземпляр `EnvironmentImpl` – класса, реализующего интерфейс `Environment`.

Чтобы вызвать методы, участвующие в сценарии, можно было бы использовать Java reflection, однако использование reflection сравнительно долгая операция, поэтому при помощи библиотеки ASM для генерации

[6]Executor в Java: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html

байткода[7] создаётся специальный класс, который вызывает $i$-ый метод из сценария для экземпляра тестируемого класса. Например, если для первого узла типа `Client` сценарий будет состоять из операций `get(2)`, `put(2, 3)`, `get(1)`, то сгенерированный код будет выглядеть следующим образом (listing 7).

Listing 7: Вызов методов из сценария

```
abstract class TestNodeExecution {
  var testInstance : Any? = null
  abstract suspend fun runOperation(i: Int) : Any?
}

class TestNodeExecution0 : TestNodeExecution {
  override suspend fun runOperation(i: Int) : Any? {
    when (i) {
      0 -> return (testInstance as Client).get(2)
      1 -> return (testInstance as Client).put(2, 1)
      2 -> return (testInstance as Client).get(1)
      else -> throw IllegalArgumentException()
    }
  }
}
```

Корутина, отвечающая за выполнение сценария, будет выглядеть примерно следующим образом (здесь опущены некоторые детали):

Listing 8: Исполнение сценария

```
for (i in 0 until scenarioSize) {
     results[iNode][i] = testNodeExecutions[iNode].runOperation(i)
     withProbability(COROUTINE_SWITCH_PROBABILITY) {
           // switch to another coroutine
           yield()
     }
}
```

Чтобы обмениваться сообщениями, используются каналы. Для каждой пары узлов создаётся корутина, которая получает и обрабатывает сообщения. Вот так выглядит корутина для узла $i$, получающая сообщения от узла $j$ (listing 9, некоторые детали опущены):

Listing 9: Получение и обработка сообщений узлом $i$ от узла $j$

```
val channel = channels[i][j]
val node = testInstances[i]
while (true) {
   val m = channel.receive()
   node.onMessage(m, j)
```

[7]https://asm.ow2.io/Библиотека для генерации байткода: https://asm.ow2.io/

```
6       withProbability(COROUTINE_SWITCH_PROBABILITY) {
7           yield()
8       }
9   }
```

`yield()` позволяет переключить исполнение на другую корутину, запущенную в том же диспатчере, если это возможно. После обработки сообщения и после окончания операции с некоторой вероятностью `COROUTINE_SWITCH_PROBABILITY` происходит переключение контекста. Таким образом поток, соотвествующий узлу, случайно переключается между выполнением операций и обработкой сообщений от других узлов, и тем самым операции и сообщения выполняются в случайном порядке.

## 4.3. Помехи в сети

В случае, если пользователь задал соответствующие настройки, требуется проэмулировать потерю, дублирование и переупорядочивание сообщений. Потеря и дублирование сообщений выглядит достаточно просто: перед отправкой делается запрос к вероятностной модели (то есть специальному классу, который генерирует значения случайных событий, происходящих в системе), чтобы определить, отправлять ли это сообщение или нет. Также в случае дублирования вероятностная модель определяет, сколько раз это сообщение будет отправлено (для того, чтобы не увеличивать время работы, сообщение может быть отправлено не более двух раз). В обоих случаях берётся фиксированная вероятность.

Для того, чтобы сообщения переупорядочивались, используется обёртка вокруг каналов со следующим интерфейсом (listing 10).

Listing 10: Интерфейс для отправки и получения сообщений

```
1 interface MessageChannel<M> {
2     fun send(m: M)
3     suspend fun receive(): M
4 }
```

В случае асинхронной доставки сообщений, получение сообщения происходит так (listing 11).

Listing 11: Канал для получения сообщения в случае отсутствия FIFO

```
class AsynchChannel<M> : MessageChannel<M> {
    ...
    val channel = Channel<M>(UNLIMITED)

    override fun send(m: M) = channel.offer(m)

    override suspend fun receive(): M {
         // suspension point
          var m = channel.receive()
          // rate in [0, MIX_RATE]
          val rate = probability.getMixRate()
          repeat(rate) {
              channel.offer(m)
              // always not null because channel is not empty
              m = channel.poll()!!
          }
          return m
    }
}
```

Сначала сообщение оказывается полученным при помощи `channel.receive()`, далее генерируется случайное число повторов `rate` от 0 до фиксированного значения `MIX_RATE`, после этого `rate` раз текущее сообщение добавляется в конец, и из канала достаётся следующее при помощи метода `poll()`, который возвращает `nullable` результат, но работает без остановки (однако в данном случае результат не будет `null`, так как в канале гарантированно что-то есть).

## 4.4. Отказы и восстановления узлов

Если алгоритм допускает отказы узлов, то эти отказы требуется искусственно добавить. Отказы могут случаться в любой точке исполнения программы, однако для упрощения они добавляются только перед или после отправки сообщения, а также перед записью в персистентное хранилище. Отказы эмулируются при помощи специального исключения, которое бросается в случае, если отказ должен произойти. При обработке этого исключения последняя операция, которая выполнялась на данном узле, помечается как отказавшая. Это означает, что при проверке на линеаризуемость будут проверяться два варианта: эта операция выполнилась полностью, или эта операция не выполнилась. После того, как узел отказал, диспатчер, соответствующий этому узлу,

переводится в состояние `CRASHED` и перестаёт выполнять оставшиеся задачи. Таким образом, действия, связанные с этим узлом, полностью прекращаются.

В случае, если поддерживается восстановление, оно происходит сразу же при обработке отказа. Создаётся новый экземпляр класса, соответствующего узлу, а также новый диспатчер, и новые каналы для передачи сообщений. После этого в новом диспатчере запускаются корутины для получения сообщений, и выполнение оставшихся операций сценария (та операция, на которой случился отказ, не перезапускается). Внутри корутины, выполняющей операции, вначале вызывается метод `recover()`. Только после этого узел помечается, как работающий, и другие узлы получают возможность отправлять ему сообщения.

Персистентное хранилище, данные из которого доступны после восстановления, представляет собой реализацию интерфейса `MutableList`, где перед операциями, меняющими его состояние, добавляется возможность отказа.

## 4.5. Расстановка отказов

В stress режиме тестирования отказы должны происходить случайно. При этом требуется, чтобы выполнялось ограничение на максимальное число отказавших узлов. Информация про число отказавших узлов должна обновляться из разных потоков. Внутри корутин нельзя использовать блокировки для синхронизации потоков. Поэтому была создана неизменяемая структура `CrashInfo`, в которой хранится информация про отказавшие узлы. Метод `crashNode(iNode: Int)` возвращает новую структуру, в которой узел с номером `iNode` помечен как отказавший, или `null`, если отказ не может быть добавлен из-за ограничений. Эта структура хранится по атомарной ссылке. Когда какой-то поток хочет добавить отказ, он заменяет структуру при помощи `compare-and-set` и возвращает `true` в случае, если отказ может быть добавлен, или возвращает `false` в случае, если из-за ограничений отказ не может быть добавлен (listing 12).

Listing 12: Обновление информации про отказы

```
while (true) {
    val prevInfo = crashInfo.value
    val newInfo = prevInfo.crashNode(iNode) ?: return false
    if (crashInfo.compareAndSet(prevInfo, newInfo)) return true
}
```

За то, произойдёт ли отказ в данный момент или нет, отвечает вероятностная модель. Контролируется математическое ожидание числа отказов, произошедших за один запуск. Чтобы этого добиться, в первый запуск отказов не происходит, а просто собирается статистика по количеству потенциальных точек отказа для каждого узла. После этого, в случае, если есть восстановление узлов, в каждой точке происходит отказ с вероятностью $E/(numOfCrashPoints * numOfNodes)$, где $E$ – это математическое ожидание числа отказов, $numOfCrashPoints$ – число потенциальных точек отказа для одного узла, а $numOfNodes$ – общее число узлов. При этом в такой модели вероятность того, что два узла откажут одновременно, становится относительно малой, а в таких ситуациях часто проявляются присутствующие в программе ошибки. Чтобы решить эту проблему, при отказе генерируется, сколько узлов подряд должны отказать (от 1 до максимального доступного числа узлов). После этого следующим $k-1$ узлам, обратившимся к вероятностной модели, автоматически возвращается, что они должны отказать ($k$ – сгенерированное случайное число отказавших подряд узлов, включая первый).

Если в случае, когда нет восстановления узлов, делать отказ с одинаковой вероятностью во всех точках, то тогда фактически вероятность отказа будет уменьшаться для каждой следующей точки, так как отказ в каждой точке случается только при условии, что во всех предыдущих точках отказа не было. Чтобы уравновесить эту разницу, требовалось определить, с какой вероятностью требуется совершать отказ в каждой точке. Для этого обозначим вероятность, с которой вероятностная модель выдаёт отказ в точке $i$ за $p_i$, а математическое ожидание числа отказов для одного узла за $q$ (заметим, что $q \leq 1$, так как узел не может отказать более одного раза). Тогда вероятность, что в точке $i$ случился

отказ равна $p_i \cdot (1 - p_{i-1}) \cdot \ldots \cdot (1 - p_1)$ и должна быть равна $q/k$, где $k$ – общее число точек, в которых возможен отказ. Тогда:

$$p_1 = \frac{q}{k}$$

$$p_2 = \frac{q}{k \cdot (1 - p_1)} = \frac{q}{k - q}$$

$$p_3 = \frac{q}{k \cdot (1 - p_2) \cdot (1 - p_1)} = \frac{q}{k \cdot \frac{k-2q}{k-q} \cdot \frac{k-q}{k}} = \frac{q}{k - 2q}$$

Таким образом, по индукции можно доказать, что $p_i = \frac{q}{k-(i-1)q}$.

$$p_i = \frac{q}{k \cdot (1 - p_{i-1}) \ldots (1 - p_1)} = \frac{q}{k \cdot \frac{k-(i-1)q}{k-(i-2)q} \ldots \frac{k-q}{k}} = \frac{q}{k - (i-1)q}$$

Эксперимент показал, что такая модель позволяет выровнить распределение отказов для одного узла. К сожалению, эта модель не учитывает, что общее число отказов для всех узлов тоже ограничено, и что если лимит на число отказов достигнут, узел также не может отказать. Однако этим обстоятельством решено было пренебречь, так как распределение от этого страдает не очень существенно. Аналогично в случае, когда есть отказы, не учитывается, что отказ в начале операции влияет на возможность отказа в конце.

В случае, если ошибка найдена, фреймворк начинает искать минимальный сценарий, при котором она воспроизводится. Для распределённых алгоритмов это включает в себя уменьшение числа отказов. Для этого математическое ожидание числа отказов постепенно уменьшается.

## 4.6. Отказы сети

Отказы сети означают, что между какими-то двумя узлами перестают доходить сообщения. Они отличаются от потери сообщений тем, что это случается на сравнительно продолжительное время. Поэтому требуется, чтобы отказы сети правильно сочетались с ограничением на максимальное количество недоступных узлов. Под недоступностью уз-

ла $A$ для узла $B$ можно понимать либо то, что сообщения, отправленные узлом $A$ узлу $B$ напрямую, не доходят, либо то, что не существует пути, по которому $A$ мог бы отправить сообщение $B$, возможно используя другие узлы. Если общее число узлов $n$, и из них могут одновременно отказать не более $x$, то в перовом случае должен существовать полный граф, в котором хотя бы $n - x$ узлов, а во втором случае – компонента сильной связности.

Инструмент поддерживает оба режима отказа сети (пользователю требуется выбрать один из них). Первый режим делит сеть на независимые компоненты, каждая из которых является полным графом. Информация про отказы сети хранится в `CrashInfo` вместе с информацией про отказавшие узлы (см. секцию 4.5) . В отличие от отказов узлов, отказы сети добавляются только перед отправкой сообщения. У вероятностной модели делается запрос, должен ли произойти отказ сети, и в случае, если получен положительный ответ, и если отказ может произойти, не нарушив ограничения, он происходит. При этом в случае, если происходит отказ единичного ребра, то отказывает ребро между отравителем сообщения и получателем, а если сеть делится на независимые компоненты, то в отказе могут быть задействованы произвольные узлы. Для восстановления сети создаётся специальная корутина, которая ждёт случайное время и после этого восстанавливает сеть.

## 4.7. Периодические таймеры

Таймеры добавляются при помощи метода `setTimer(name, ticks, f)` у класса `Environment`. Имя таймера сохраняется, а для исполнения в диспатчер, соответствующий данному узлу, запускается следующая корутина (listing 13).

Listing 13: Корутина, выполняющая периодические операции таймера

```
while(!isFinished && timers.contains(name)) {
    f()
    // suspends
    delay(ticks * TICK_TIME)
}
```

В цикле проверяется, что исполнение не завершилось, и что этот таймер не удалили, после этого выполняется функция `f()`, а потом `delay()` приостанавливает исполнение на время $t = ticks \cdot TICK\_TIME$ миллисекунд. Таким образом, с интервалом $t$ в диспатчер, в котором была запущена эта корутина, добавляется задача "проверить, работает ли таймер, и если да, то выполнить действие этого таймера".

На текущий момент `TICK_TIME` – это константа, которая также используется для таймаутов (см. 4.8). В будущем планируется подбирать её значение так, чтобы исполнение алгоритма работало как можно быстрее.

## 4.8. Определение окончания исполнения

Требовалось определить, когда исполнение должно быть завершено. К этому могут быть два подхода. Первый способ подразумевает, что исполнение завершается тогда, когда для всех операций получены результаты, несмотря на то, что некоторые сообщения остались необработанными. Второй подход подразумевает, что исполнение останавливается, когда в системе больше ничего не происходит. Проблемы первого подхода заключаются в том, что:

1. Когда в конце дополнительно проверяются какие-то инварианты при помощи функций для валидации. В такой ситуации может быть критично, чтобы все отправленные сообщения были обработаны, так как иначе инварианты могут не выполняться.

2. Если операция может блокироваться, то есть существует валидный сценарий, при котором операция никогда не завершится. Например, в случае алгоритма распределённой блокировки, если два узла попытаются одновременно взять блокировку и не будут отпускать её, то один из этих узлов не должен преуспеть.

Второй же подход плох тем, что в системе есть таймеры, которые продолжают свою работу до тех пор, пока алгоритм не выполнился. Это влечёт за собой то, что какие-то активные действия узлов могут

быть вызваны срабатыванием таймера. То есть если все операции в системе приостановлены, никаких сообщений нет, но работают таймеры, нет гарантии, что на какое-то $n$-ое срабатывание таймера узлы начнут отправлять друг другу сообщения, которые приведут к тому, что все операции корректно завершатся.

Чтобы решить эту проблему, было предложено следующее: если в системе есть блокирующиеся операции, это указывается в аннотации к этим операциям, и тогда считается, что операции не могут возобновляться при помощи таймеров. Тогда исполнение завершается, когда не осталось активных корутин. Если же таких операций нет, то исполнение завершается, когда не осталось активных корутин, и все операции завершились (то есть когда выполнились оба условия).

Для того, чтобы определить, что больше не осталось активных корутин, заводится общий счётчик задач на все диспатчеры, который увеличивается, когда задача добавляется в диспатчер, и уменьшается, когда задача выполнена. Когда счётчик становится равным 0, исполнение завершается. При этом дополнительно счётчик увеличивается в начале операции и уменьшается в конце, что позволяет не завершиться до того момента, как операция была выполнена. Более подробно общая схема происходящего выглядит так:

1. Счётчик инициализируется числом начальных задач. Начальными задачами считаются исполнение операций из сценария и получение сообщения между каждой упорядоченной парой узлов. Это делается для того, чтобы счётчик не стал равным 0 прежде, чем все начальные задачи были добавлены. При этом требуется уметь отличать начальные задачи от всех остальных, чтобы при добавлении начальной задачи счётчик не увеличивался, так как эти задачи уже были учтены. Для этого используется корутинный контекст, который передаётся в диспатчер вместе с задачей, которую нужно исполнить.

2. В случае отказа соответствующего узла диспатчер переходит в состояние `CRASHED`. Это означает, что он перестаёт выполнять зада-

чи, а лишь увеличивает и уменьшает счётчик для всех оставшихся и добавленных задач.

3. В `Environment` есть функция `withTimeout`, которая гарантированно продолжает исполнение после прошествия фиксированного времени. Использование этой функции подразумевает то, что исполнение не должно завершиться, пока эта функция не сработала. Поэтому счётчик увеличивается в начале операции и уменьшается в конце. Аналогично происходит при выполнении операций, если они не помечены как блокирующиеся. Однако, если в процессе ожидания узел отказал, то блок, в котором счётчик уменьшается, никогда не будет выполнен, и счётчик никогда не будет равен 0. Для того, чтобы этого не случилось, каждый диспатчер хранит в себе локальную переменную, в которую он сохраняет все дополнительные увеличения счётчика, и в случае отказа значение этой переменной вычитается из общего счётчика.

## 4.9. Выводы и результаты главы

Был поддержан режим stress тестирования для распределённых алгоритмов. Для ускорения тестирования было решено не создавать отдельные процессы для узлов. В stress режиме тестирования каждый узел работает в своём потоке, происходит много запусков, и за счёт случайного переключения контекста ошибка находится с высокой вероятностью. Для получения сообщений, для выполнения операций из сценария и для работы периодических таймеров используются корутины. Отказы в систему добавляются случайно. Исполнение заканчивается тогда, когда не осталось активных корутин, и, в случае, если в сценарии нет блокирующихся операций, когда все операции завершены.

# 5. Model checking

В настоящий момент создан лишь прототип model checking, который поддерживает не все опции, поддерживаемые stress режимом. Тем не менее, текущую версию model checking уже можно использовать для тестирования алгоритмов.

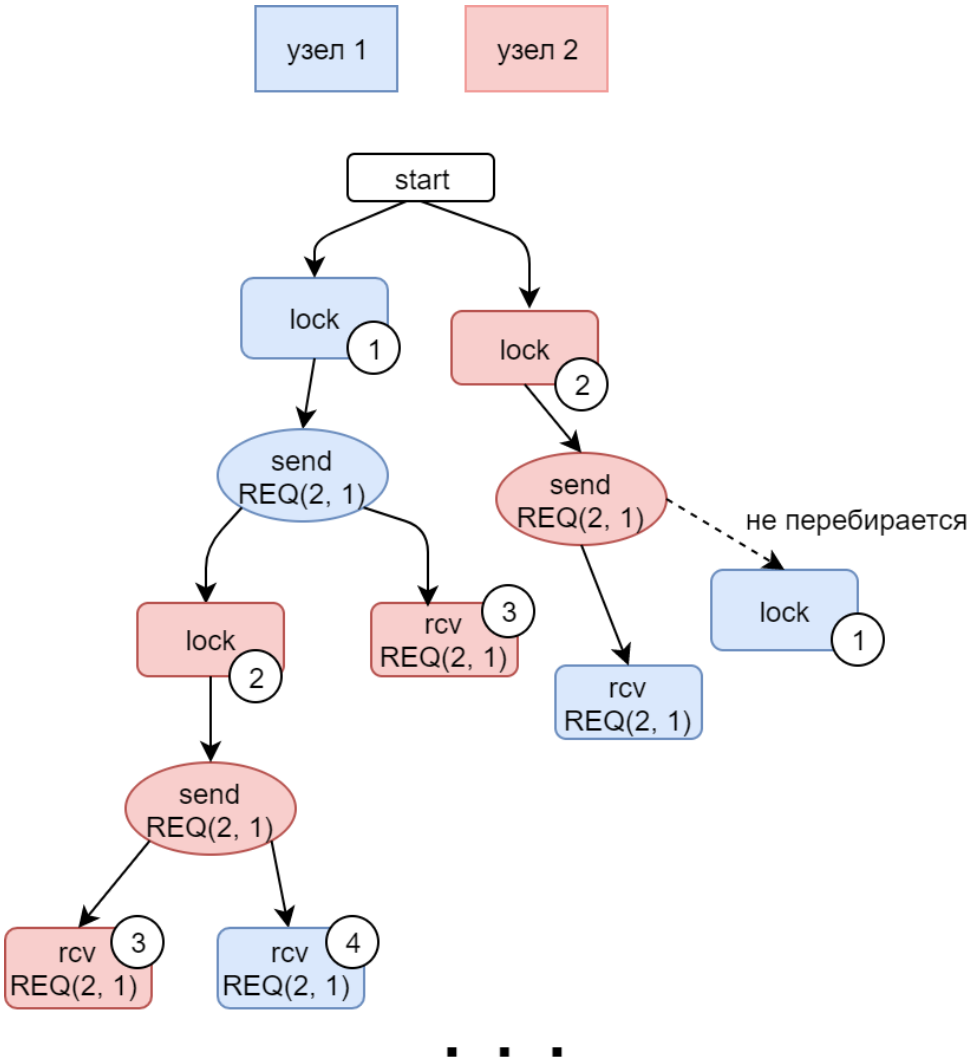
## 5.1. Общая идея

Model checking – это проверка свойств системы путём исчерпывающего перебора. Bounded model checking предполагает исчерпывающий перебор всех вариантов исполнения с каким-то заданным ограничением. Например, в случае model checking для многопоточных структур в Lincheck ограничивалось число переключений контекста (то есть переключений между потоками). Позаимствовать эту идею напрямую в распределённые алгоритмы было нельзя, так как, во-первых, в отличие от многопоточных алгоритмов, для того, чтобы выполнить операцию, требуется постоянно переключаться между узлами, и поэтому минимизировать число переключений будет нецелесообразно, а во-вторых, для одного узла может существовать много разных задач (например, обработать сообщения от разных узлов), и непонятно, какую из этих задач требуется выполнить первой. Чтобы адаптировать этот подход на распределённые алгоритмы, требовалось придумать аналог переключений контекста. Для этого требовалось выбрать один базовый вариант исполнения, при котором значение ограничиваемого параметра было бы равно 0. Под этим вариантом было решено выбрать исполнение, когда задачи (операция или обработка сообщений) выполняются в порядке их возникновения. Ограничиваться будет число задач, исполненных раньше своей очереди. При этом очередь задач – общая на все узлы в системе, и задачи будут упорядочены даже если они независимы и относятся к разным узлам. Также если в системе возможны отказы, то число отказов также должно входить в ограничиваемый параметр. При этом хочется уменьшить число перебираемых вариантов исполнения, по возможности не перебирая эквивалентные варианты.

## 5.2. Пример дерева исполнения для алгоритма Лампорта распределённой блокировки

Рассмотрим на примере, какие могут быть варианты исполнения для алгоритма Лампорта распределённой блокировки [7] для случая двух узлов, каждый из которых выполняет операцию `lock`.

Рис. 2: Пример дерева исполнений для алгоритма Лампорта для распределённой блокировки. Цифры обозначают порядок, в котором задачи появлялись. Пунктиром показывается порядок исполнения, который эквивалентен другому порядку исполнения, и который поэтому не имеет смысла перебирать




. . .

В начале исполнения у нас есть выбор из двух вариантов: начать операцию на первом узле или начать операцию на втором узле. Будем

нумеровать все возникающие по ходу исполнения задачи, такие как начало операции или обработка сообщения. При выполнении операции на первом узле узлу 2 будет отправлено сообщение $REQ(2, 1)$ с запросом на взятие блокировки. После этого в дерево добавляется новая задача для второго узла: обработать это сообщение. Эта задача будет идти под номером 3. Теперь у нас есть выбор: начать операцию на втором узле (задача 2) или обработать сообщение $REQ$ от первого узла (задача 3). Если выбрать задачу 2, то узел 2 отправит запрос на взятие блокировки узлу 1, и появится новая задача под номером 4 для первого узла по обработке этого сообщения. После этого будет выбор между задачами 3 и задачами 4, и так далее. При этом если в дальнейшем продолжить перебор всевозможных вариантов и начать с операции на втором узле, то есть с задачи 2, то после этого не имеет смысла выполнять задачу 1, так как это будет эквивалентно варианту, когда сначала была выполнена задача 1, а после этого задача 2, а такой вариант уже был рассмотрен.

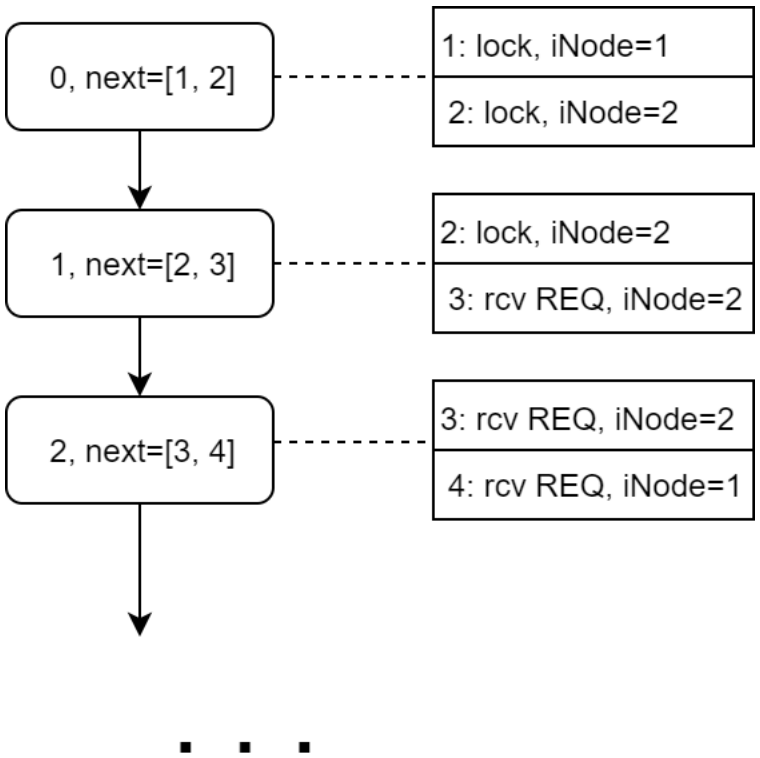
## 5.3. Реализация режима model checking

Чтобы систематически исследовать всевозможные варианты исполнения, требуется хранить информацию об уже исследованных вариантах. Для этого строится дерево интерливингов (интерливинг – один фиксированный порядок исполнения событий).

По ходу исполнения все возникающие задачи (начало новой операции или обработка сообщений) не выполняются сразу, а нумеруются при помощи автоматически увеличивающегося счётчика и добавляются в специальную очередь. На каждом шаге выбирается следующая задача, которая должна быть выполнена. Параллельно происходит спуск по дереву, которое хранит в себе информацию об исследованных интерливингах. Путь вниз по дереву соответствует какому-то интерливингу. В каждом узле хранится номер задачи из очереди, которая выполняется в данный момент, номера задач из очереди, которые можно было бы выполнить после этой задачи, а также ссылки на следующие, уже

исследованные узлы. Например, в случае описанного выше примера с алгоритмом распределённой блокировки 2, для первой итерации, когда ограничиваемый параметр был равен 0, то есть все задачи выполнялись в порядке их возникновения, дерево исполнений вместе с очередью задач на каждом переходе будет выглядеть как на рисунке 3.

Рис. 3: Как строится дерево исполнения в нашей реализации для примера 2. Слева показаны узлы дерева. Так как это первая итерация, то пока у каждого узла есть только одна дочерняя вершина. В каждой вершине хранится номер задачи, которая выполнялась в в момент посещения этой вершины. Помимо этого у каждой вершины хранится список номеров задач, которые можно было выполнить после неё (next). Справа от каждого узла указано, какие задачи были в очереди на момент завершения соответствующей узлу задачи. У каждой задачи есть номер, действие, которое надо выполнить, и номер узла, к которому эта задача относится. Из этой очереди узел сохраняет себе значения возможных переходов.




. . .

После того, как была собрана информация про возможные переходы, на каждой итерации строится путь, по которому должно идти исполнение в следующий раз, с учётом ограничиваемого параметра. Он

заканчивается на переходе в вершину, которая ещё не была изучена, и которой пока нет в дереве исполнений. Например, для дерева исполнений на рисунке 3 путь мог бы выглядеть так: $0 \to 1 \to 3$. После того, как была бы выполнена задача 3, путь, который был построен, закончился, и дальше для того, чтобы завершить исполнение, для каждой вершины будет выбираться задача с наименьшим номером, в которую есть переход. После того, как исследованы все варианты со значением параметра $k$, значение параметра увеличивается на 1. При построении пути, по которому будет проходить исполнение, следующая вершина выбирается при помощи взвешенного рандома. Вес вершины соответствует доле исследованных путей из неё. Если вершина полностью исследована, то в неё переход осуществляться не будет.

Чтобы перебирать меньше эквивалентных вариантов, переход делается не во все возможные вершины, а лишь в те, у которых номер больше, либо которые относятся к тому же узлу. Докажем от противного, что если делать только такие переходы, то не будет интерливинга, который будет пропущен. Пусть не так, тогда есть интерливинги, в которых это условие нарушается, и которые не эквивалентны ни одному интерливингу, в котором бы это условие не нарушалось. Среди всех таких интерливингов рассмотрим тот, в котором нарушение этого условия в первый раз происходит на самой большой позиции среди всех. Пусть первое нарушение этого условия в этом интерливинге происходит, когда из задачи с номером $x$ и номером узла $i$ случился переход в задачу с номером $y$ и с номером узла $j$, при этом известно, что $x > y \wedge i \neq j$. Тогда рассмотрим интерливинг, в котором задачи $x$ и $y$ будут поменяны местами. Этот интерливинг является валидным, так как задача $y$ возникла раньше, чем задача $x$, так как она имеет меньший номер, то есть на момент начала выполнения задачи $x$ она уже существовала, и её выполнение не могло нарушать никакие ограничения, так как единственным возможным ограничением на порядок выполнения задач является требования FIFO порядка для доставки сообщений, но задача $y$ была создана раньше задачи $x$, и к тому же эти задачи выполняются

на разных узлах. При этом этот интерливинг будет эквивалентен исходному, так как до выполнения этих задач состояние системы было одинаковым, а выполнение каждой задачи зависит только от состояния узла, на котором она выполняется, значит, так как узлы разные, итоговое состояние системы будет одинаковым вне зависимости от порядка выполнения этих операций. Если после этого задача, которая раньше стояла до $x$, и задача $y$ снова упорядочены неправильно, повторим эту процедуру и продолжим это делать до тех, пор пока $y$ не станет либо первой задачей в интерливинге, либо правильно упорядочена относительно предыдущей задачи. Таким образом можно построить интерливинг, который эквивалентен исходному, и в котором это условие либо не нарушается, либо первое нарушение происходит на строго большей позиции, а это противоречит нашему выбору.

При этом может возникнуть ситуация, когда невозможно сделать переход, который бы не нарушал это условие, а в очереди ещё остались бы задачи. Тогда будем подниматься по дереву до тех пор, пока не придём в вершину, из которой можно сделать переход в одну из задач из очереди. После этого ветка, по которой был сделан переход из этой вершины, выкидывается и удаляется из множества возможных переходов. Докажем, что это корректно. Пусть вершина $A$ – это первая вершина, в которую можно сделать переход в задачу из очереди $C$, а $B_1 \to B_2 \to \ldots \to B_k$, это путь, по которому происходил спуск из $A$. Известно, что $\forall i \in 1..k\ id(B_i) > id(C) \wedge node(B_i) \neq node(C)$. Это означает, что порядок исполнения $A \to C \to B_1 \to \ldots \to B_k$ будет эквивалентен $A \to B_1 \to \ldots \to B_k \to C$, при этом в первом случае в этой ветке не будут нарушаться условия перехода.

Таким образом, эквивалентные варианты всё же иногда перебираются, но зато они сразу же удаляются из дерева и не хранятся в памяти. Это важно, так как режим model checking работает быстрее, чем stress режим (см. таблицу 3), поэтому время работы не требовалось существенно оптимизировать, однако расход памяти является узким местом.

Отказы в системе также соответствуют вершинам дерева и задачам

из очереди. Однако в отличие от других задач они добавляются в очередь только в случае, когда отказ должен случиться в этом месте, а при обычном исполнении в точках, в которых может произойти отказ, счётчик, нумерующий отказы, просто увеличивается, чтобы присвоить потенциальным отказам уникальный идентификатор.

## 5.4. Выводы и результаты главы

Был реализован режим bounded model checking для распределённых алгоритмов. В нём выбирается базовый вариант исполнения, в котором события обрабатываются в порядке своего возникновения. Ограничиваемым параметром является общее число инверсий относительно этого порядка и отказов. В процессе строится дерево интерливингов, хранящее в себе информацию об уже исследованных вариантах исполнения. Добавлена простая оптимизация, позволяющая перебирать меньшее число эквивалентных вариантов и не хранить их в памяти. На текущий момент из-за нехватки времени режим model checking не поддерживает потерю и дублирование сообщений, разделение сети, а также возможность устанавливать периодические таймеры и таймауты.

# 6. Тестирование полученного инструмента

Требовалось убедиться, что инструмент применим к широкому классу алгоритмов. Для этого с его помощью были реализованы и протестированы различные известные алгоритмы для распределённых систем. Часть из них тестировалась при помощи проверки результатов на линеаризуемость, часть – при помощи функций для валидации, которым на вход передавался список событий, происходивших в системе во время исполнения.

Таблица 2: Распределённые алгоритмы, реализованные с помощью Lincheck

| **Категория** | **Алгоритмы** |
|---|---|
| Распределённое хранилище вида "ключ-значение" | Exactly-once execution в случае наличия помех в сети<br>Репликация данных в случае отсутствия отказов узлов<br>Алгоритм Raft[12] |
| Алгоритмы взятия распределённой блокировки | Алгоритм Лампорта для распределённой блокировки [7]<br>Алгоритм Рикарта-Агравалы [16]<br>Централизованный алгоритм |
| Алгоритмы широковещательной рассылки | Алгоритм Скина [4]<br>Широковещательная рассылка в случае отказов узлов |
| Алгоритм получения глобального состояния системы | Алгоритм Чанди-Лампорта [1] |
| Алгоритм распределённого консенсуса | Алгоритм распределённого консенсуса в случае отсутствия отказов<br>Алгоритм Raft (без репликации) |

Чтобы убедиться в эффективности алгоритма в поиске ошибок, были созданы "наивные реализации" тестируемых алгоритмов, а также версии алгоритмов с намеренно добавленными в них ошибками. Например, в алгоритме Raft узел не сохранял персистентно значение переменной `votedFor`, обозначающей, за кого узел отдал свой голос. В таком случае, если узел отказывал и восстанавливался, он мог дважды проголосовать в одном раунде за разные узлы, что приводило к тому, что в системе появлялось два лидера. Помимо этого, корректные алгоритмы тестировались в условиях, на которые они не были рассчитаны. Например, алгоритм Лампорта для распределённой блокировки перестаёт корректно работать, если в системе нет FIFO порядка. Во всех случаях ошибки были найдены.

Таблица 3: Время тестирования различных алгоритмов

| **Алгоритм** | **Число итераций** | **Число запусков** | **Время работы, stress** | **Время поиска ошибки stress** | **Время работы, mc** | **Время поиска ошибки, mc** |
|---|---|---|---|---|---|---|
| Lamport mutex | 10 | 3000 | 51s 238mc | 4s 447mc | 31s 133mc | 9s 198 mc |
| Ricart-Agrawala | 10 | 3000 | 49s 999mc | 1s 223mc | 27s 764mc | 1s 190 mc |
| Chandy-Lamport | 10 | 3000 | 53s 924mc | 7s 690mc | 37s 648mc | 10s 909 mc |
| Skeen | 1 | 30000 | 47s 621mc | 1s 161mc | 1min 19s | 1s 207mc |
| Broadcast with crashes | 1 | 30000 | 53s 643mc | 4s 865mc | 29s 690mc | 10s 51mc |
| Raft | 10 | 1000 | 9min 55s | 21s 592mc | - | - |

Эксперименты проводились на компьтере Acer Swift SF713-51 2017 года выпуска, имеющем 4-ядерный процессор Intel Core i7 1.30ГГц и 8Гб оперативной памяти.

В таблице 3 приведено время работы некоторых из реализованных алгоритмов в режиме stress и model checking (mc), а также среднее время поиска ошибки. В алгоритмах участвовало 3 узла, у каждого узла было по 3 операции в сценарии. Для большинства алгоритмов stress режим и model checking режим работают в пределах 1-2 минут

для 3000 запусков в совокупности. Исключение состовляет алгоритм Raft, который работает 10 минут для 1000 запусков. Это происходит из-за того, что алгоритм Raft использует случайные таймауты для того, чтобы два узла не начали выборы нового лидера одновременно, и это сильно замедляет время работы программы. В дальнейшем планируется попытаться оптимизировать работу с таймерами и таймаутами. Протестировать алгоритм Raft в режиме model checking на данный момент невозможно, так как в режиме model checking не поддерживаются таймеры.

## 6.1. Выводы и результаты главы

Для того, чтобы проверить эффективность инструмента, с его помощью были реализованы различные распределённые алгоритмы. В некоторые реализации были намерено добавлены ошибки. Помимо этого алгоритмы тестировались в условиях, на которые они не были рассчитаны. Во всех случаях, когда инструмент должен был находить ошибку, ошибка была найдена. Также показано, что тестирование происходит за разумное время, что позволяет использовать инструмент на практике.

# Заключение

В рамках этой работы в Lincheck была добавлена возможность тестирования распределённых алгоритмов при помощи эмуляции распределённой системы. В частности, был разработан программный интерфейс, позволяющий пользователю реализовывать распределённые алгоритмы, были поддержаны различные типы отказов узлов и сети, и было реализовано тестирование в stress и model checking режимах.

Инструмент протестирован на различных распределённых алгоритмах, которые были реализованы с его помощью. Корректные реализации успешно проходят проверку, а в некорректных инструмент находит ошибку.

Дальнейшая работа возможна в следующих направлениях:

1. Поддержка всей функциональности для режима model checking, которая поддерживается в stress режиме. Помимо этого, в model checking можно ещё сократить перебор эквивалентных вариантов, например воспользовавшись техникой dynamic partial order reduction.

2. Оптимизация времени работы для stress режима. Это можно сделать за счёт подбора оптимального значения единицы времени для таймеров и таймаутов в процессе исполнения и за счёт собственной реализации каналов, позволяющих передавать сообщения быстрее.

3. Поддержка других моделей согласованности помимо линеаризуемости.

4. Создание плагина для Intellij IDEA для визуализации распределённых алгоритмов. Это может помочь упростить отладку распределённых алгоритмов.

5. Поддержка других параметров распределённых систем, например, возможности динамического добавления узлов в систему.

# Список литературы


[1] Chandy K Mani, Lamport Leslie. Distributed snapshots: Determining global states of distributed systems // ACM Transactions on Computer Systems (TOCS). — 1985. — Vol. 3, no. 1. — P. 63–75.

[2] Flanagan Cormac, Godefroid Patrice. Dynamic partial-order reduction for model checking software // ACM Sigplan Notices. — 2005. — Vol. 40, no. 1. — P. 110–121.

[3] Flymc: Highly scalable testing of complex interleavings in distributed systems / Jeffrey F Lukman, Huan Ke, Cesar A Stuardo et al. // Proceedings of the Fourteenth EuroSys Conference 2019. — 2019. — P. 1–16.

[4] Garg Vijay K. Total order multiocast // Concurrent and distributed computing in Java. — IEEE Press, 2005.

[5] JEPSEN. — Access mode: `https://jepsen.io/`.

[6] Lamport Leslie. The temporal logic of actions // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1994. — Vol. 16, no. 3. — P. 872–923.

[7] Lamport Leslie. Time, clocks, and the ordering of events in a distributed system // Concurrency: the Works of Leslie Lamport. — 2019. — P. 179–196.

[8] Lamport Leslie et al. Paxos made simple // ACM Sigact News. — 2001. — Vol. 32, no. 4. — P. 18–25.

[9] Life, death, and the critical transition: Finding liveness bugs in systems code / Charles Killian, James W Anderson, Ranjit Jhala, Amin Vahdat / NSDI. — 2007.

[10] MODIST: Transparent model checking of unmodified distributed systems / Junfeng Yang, Tisheng Chen, Ming Wu et al. — 2009.

[11] Maelstorm. — Access mode: https://github.com/jepsen-io/maelstrom.

[12] Ongaro Diego, Ousterhout John. In search of an understandable consensus algorithm // 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). — 2014. — P. 305–319.

[13] Panda Aurojit, Sagiv Mooly, Shenker Scott. Verification in the age of microservices // Proceedings of the 16th Workshop on Hot Topics in Operating Systems. — 2017. — P. 30–36.

[14] Practical software model checking via dynamic interface reduction / Huayang Guo, Ming Wu, Lidong Zhou et al. // Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. — 2011. — P. 265–278.

[15] Ozkan BK, Majumdar R, Niksic F et al. Randomized testing of distributed systems with probabilistic guarantees. PACMPL 2 (OOPSLA), 160: 1–160: 28 (2018).

[16] Ricart Glenn, Agrawala Ashok K. An optimal algorithm for mutual exclusion in computer networks // Communications of the ACM. — 1981. — Vol. 24, no. 1. — P. 9–17.

[17] {SAMC}: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems / Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi et al. // 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). — 2014. — P. 399–414.

[18] Teaching rigorous distributed systems with efficient model checking / Ellis Michael, Doug Woos, Thomas Anderson et al. // Proceedings of the Fourteenth EuroSys Conference 2019. — 2019. — P. 1–15.

[19] Testing concurrency on the JVM with lincheck / Nikita Koval, Maria Sokolova, Alexander Fedorov et al. // Proceedings of the 25th

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2020. — P. 423–424.

[20] Use of formal methods at Amazon Web Services / Chris Newcombe, Tim Rath, Fan Zhang et al. // See http://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf. — 2014.

[21] Viotti Paolo, Vukolić Marko. Consistency in non-transactional distributed storage systems // ACM Computing Surveys (CSUR). — 2016. — Vol. 49, no. 1. — P. 1–34.

[22] What bugs live in the cloud? a study of 3000+ issues in cloud systems / Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa et al. // Proceedings of the ACM Symposium on Cloud Computing. — 2014. — P. 1–14.

[23] Yuan Xinhao, Yang Junfeng. Effective Concurrency Testing for Distributed Systems // Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. — 2020. — P. 1141–1156.

[24] Yuan Xinhao, Yang Junfeng, Gu Ronghui. Partial order aware concurrency sampling // International Conference on Computer Aided Verification / Springer. — 2018. — P. 317–335.

[25] A formally verified nat / Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa et al. // Proceedings of the conference of the acm special interest group on data communication. — 2017. — P. 141–154.