Декларативное тестирование высокоуровневых свойств распределённых алгоритмов

Лупуляк Ольга Сергеевна

научный руководитель: Коваль Никита Дмитриевич

СПб ВШЭ

11 июня 2021 г.

Мотивация

- В процессе обучения или при прототипировании распределённые алгоритмы полезно реализовывать отдельно, не тратя силы на технические сложности, связанные с реализацией полноценной распределённой системы
- Тестирование распределённых систем – сложно, так как существует большое число вариантов исполнения, особенно для случаев, когда надо тестировать отказоустойчивость
- Хотелось бы иметь инструмент, позволяющий как можно проще реализовывать и тестировать распределённые алгоритмы

Существующие инструменты

Название	**Метод**	**Область применения**
MaceMC [1], SAMC [2], MODIST [3]	Model checking	Теоретические работы
Jepsen [4]	Стресс тестирование	Black-box тестирование промышленных распределённых систем
Morpheus [5]	Случайное тестирование с эвристиками	Распределённые системы на языке Erlang
DEMi [6]	Стресс тестирование с минимизацией ошибки	Распределённые системы, основанные на Akka
DSLabs [7]	Model checking	Проверка домашних работ

Сложность использования существующих инструментов

Название	**Размер теста**	**Язык программирования**
Jepsen	$100 - 1000$ строк кода	Clojure
Morpheus	≈ 600 строк кода	Erlang
DEMi	≈ 500 строк кода	Scala
DSLabs	≈ 1000 строк кода	Java

Lincheck

```
1 @StressCTest
2 class StackTest {
3     val stack = Stack<Int>()
4     @Operation fun pop(): Int = stack.pop()
5     @Operation fun push(value: Int) = stack.push(value)
6
7     @Test fun test() = Linchecker.check(this::class)
8 }
```

- Тестирование многопоточных структур данных под JVM
- Генерация сценария
- Запуск нескольких итераций
- Проверка результатов - поиск последовательной истории
- Работает в стресс и model checking режимах

Распределённые и многопоточные алгоритмы

Общее:

- Операции выполняются параллельно
- Похожие критерии корректности

Различия:

- В распределённых алгоритмах узлы могут быть разнесены в пространстве
- Обмен сообщениями вместо общей памяти
- Возможность различных отказов
- Операция чаще всего выполняется на нескольких узлах

Цель и задачи

Цель - расширить Lincheck, добавив возможность тестирования распределённых алгоритмов.

Задачи:

- Разработать программный интерфейс для реализации и тестирования распределённых алгоритмов с помощью Lincheck
- Реализовать стресс режим тестирования распределённых алгоритмов
- Адаптировать model checking под распределённые алгоритмы
- Протестировать полученный инструмент на существующих алгоритмах

Параметры конфигурации для распределённых систем

Поддерживается:

- Типы участвующих в алгоритме узлов
- Ограничения на число узлов каждого типа
- Отказы узлов и сети (потеря сообщений, дублирование сообщений, отсутствие FIFO, отказ узла с восстановлением и без него, разделение сети)
- Максимальное число одновременно недоступных узлов

Пример теста

В системе от 1 до 3 клиентов и от 1 до 5 серверов. Могут происходить разделения сети и отказы с восстановлением. Одновременно могут стать недоступны менее половины серверов.

```
1  @Test
2  fun test() =
3      DistributedOptions()
4          .sequentialSpecification(HashMap::class.java)
5          .nodeType(Server::class.java, 5)
6          .nodeType(Client::class.java, 3)
7          .networkPartitions(HALVES)
8          .maxNumOfFailedNodes(Server::class.java) { n ->
9              (n - 1) / 2
10         }
11         .crashMode(RECOVERY)
12         .check()
```

Операции в распределённых алгоритмах

- Проверяемые операции – это методы класса, обозначающего узел, по аналогии с обычной структурой данных
- Большинство операций в распределённых алгоритмах асинхронны из-за необходимости взаимодействия с другими узлами
- Внутри одного узла присутствуют разные задачи: выполнение операций из сценария, обработка сообщений из других узлов, срабатывание периодических таймеров.
- Все задачи узла должны выполняться в одном потоке, чтобы пользователю не надо было заботиться о потокобезопасности

Решение – использовать корутины

Пример реализации узла

```
1  class Client(val env: Environment) : Node<Message> {
2      // get server address
3      val server = env.getAdressesForClass(Server::class.java)[0]
4      ...
5      // called when message is received
6      override fun onMessage(msg: Message, sender: Int) {
7          when (msg) {
8              is GetResponse -> {
9                  result = msg.result
10                 // signals to resume operation
11                 semaphore.signal()
12             }
13             ...
14         }
15     }
16
17     @Operation
18     suspend fun get(key: String) : String? {
19         env.send(GetRequest(key), server)
20         // awaits for result
21         semaphore.await()
22         return result
23     }
24 }
```

Стресс режим

1. При помощи байткода генерируется вызов методов из сценария
2. Для каждого узла запускается корутина для выполнения сценария и корутины для получения и обработки сообщений
3. Сообщения передаются при помощи каналов
4. Исполнение заканчивается когда не остаётся активных корутин

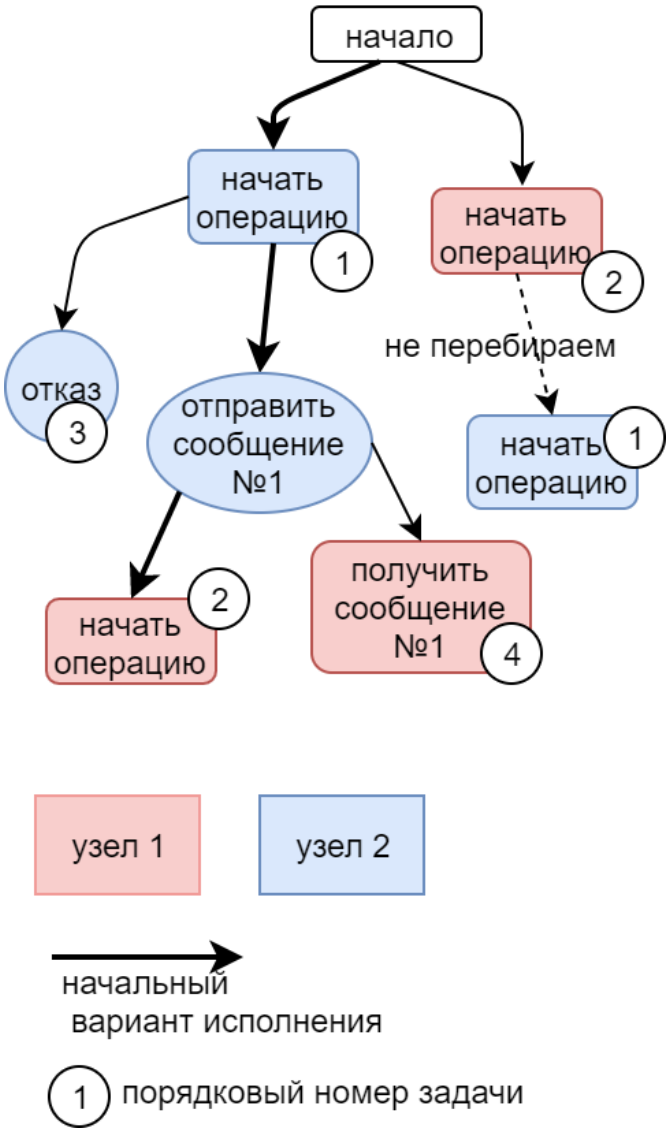
Получение сообщений узлом i от узла j

```
1 val node = nodes[i]
2 while(true) {
3     val message = channel.receive()
4     node.onMessage(message, j)
5 }
```

Отказы узлов

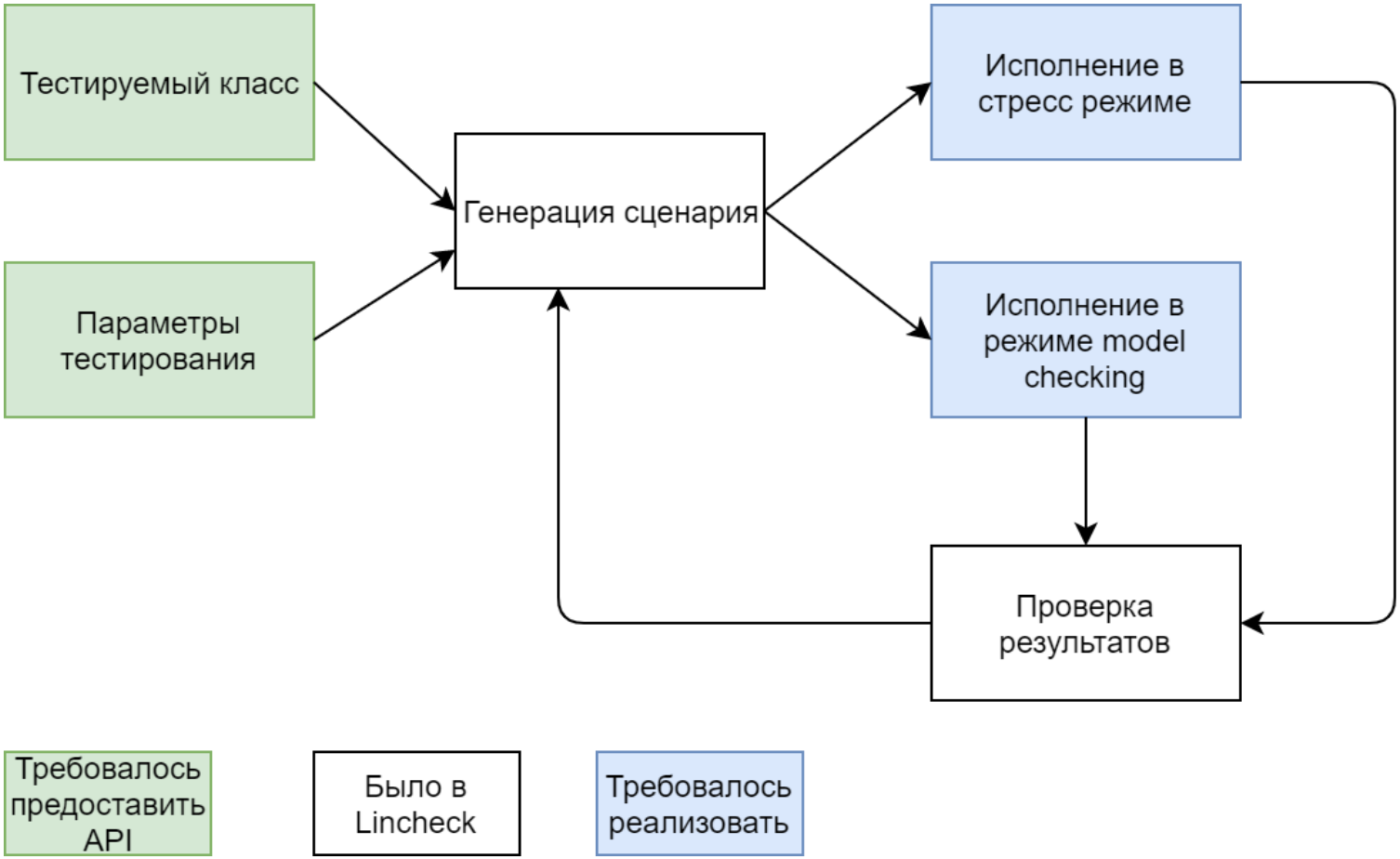
- Отказ происходит перед или после отправки сообщения или перед записью в персистентное хранилище
- Информация про отказавшие узлы хранится в неизменяемой структуре, которая подменяется при помощи `compare-and-set`
- Отказы происходят случайно, контролируется матожидание числа отказов
- В случае отказа кидается специальное исключение
- Оставшиеся для этого узла задачи не выполняются
- При восстановлении новый экземпляр класса и новые каналы

Model checking



- Начальный порядок исполнения – задачи выполняются в порядке их возникновения
- Ограничиваемый параметр – общее число отказов и инверсий относительно начального варианта исполнения
- Не перебираются эквивалентные варианты

Изменения, которые требовалось сделать в Lincheck



Алгоритмы, на которых тестировался инструмент

Полученный инструмент позволяет тестировать различные классы алгоритмов, например:

- Распределённое хранилище вида ”ключ-значение”
- Алгоритмы распределённой блокировки (алгоритм Лампорта, алгоритм Рикарда-Аргвалы, централизованный алгоритм)
- Общий порядок для широковещательной рассылки (алгоритм Скина)
- Алгоритм получения глобального состояния (алгоритм Чанди-Лампорта)
- Алгоритмы распределённого консенсуса (Raft)

Результаты

- Разработано API для реализации распределённых алгоритмов внутри Lincheck
- Реализован стресс режим для тестирования
- Реализован model checking
- Инструмент проверен на реализациях известных алгоритмов, содержащих намеренно допущенные ошибки

Ссылки

- [1] Killian, C., Anderson, J. W., Jhala, R., Vahdat, A. (2007). Life, death, and the critical transition: finding liveness bugs in systems code.
- [2] Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J. F. (2014). SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems.
- [3] Junfeng Yang, Tisheng Chen, Ming Wu et al. (2009). MODIST: Transparent model checking of unmodified distributed systems
- [4] Distributed Systems Safety Research. (2016). Jepsen. https://jepsen.io

Ссылки

- [5] Yuan Xinhao, Yang Junfeng. (2020). Effective Concurrency Testing for Distributed Systems
- [6] Scott, C., Brajkovic, V., Necula, G., Krishnamurthy, A., Shenker, S. (2016). Minimizing faulty executions of distributed systems.
- [7] Michael, E., Woos, D., Anderson, T., Ernst, M. D., Tatlock, Z. (2019). Teaching Rigorous Distributed Systems With Efficient Model Checking. https://github.com/emichael/dslabs

Генерируемые отказы

Тип отказа	Режимы	Определение доступности
Отказ сети	Между любыми двумя узлами	Существует компонента сильной связности (*)
	Разделение сети	Существует полный граф (*)
Отказ узла	С восстановлением	Зависит от режима отказа сети
	Без восстановления	
Потеря сообщений		Никак
Дублирование сообщений		-
Отсутствие FIFO		-

Ограничение на число недоступных одновременно узлов (*):
$\forall$ типа узла i если n_i – число узлов этого типа, а x_i – максимальное число недоступных узлов этого типа, должно присутствовать $n_i - x_i$ узлов типа i

Ограничения

- Нет масштабируемости
- Нельзя проверять производительность
- Не проверяются модели согласованности помимо линеаризуемости
- Не поддерживается сложная топология
- Узлы не могут добавляться по мере исполнения
- Пока плохо эмулируются задержки сообщений
- Генерация помех в сети для конкретных типов узлов

Что проверяется

- Линеаризуемость результатов операций
- Выпадающие исключения
- Что исполнение завершается за конечное время
- Произвольные инварианты, определяемые пользователем

Корутины

- Легковесные потоки для асинхронных задач
- Пристанавливаются в специальных точках
- Диспатчер корутин для выполнения задач от одной точки остановки до другой

Определение окончания исполнения

- Для каждого узла создаётся свой диспатчер
- Считается общая сумма всех задач в диспатчерах
- При отказе оставшиеся в диспатчере задачи для узла перестают выполняться, и меняется только счётчкик
- Для `suspend` функций, возобновляющихся через какое-то время, вроде `delay`, счётчик дополнительно увеличивается в начале, и уменьшается в конце
- Чтобы учесть такие дополнительные увеличения счётчиков в случае отказа, информация про них хранится в локальной переменной для каждого диспатчера

Реализация таймеров

```
1 launch(dispatcher) {
2    while(!isFinished) {
3        f()
4        delay(ticks)
5    }
6 }
```

- Работает до тех пор, пока происходит исполнение
- Проблема: если операция сценария ждёт события, связанного с таймерами, то программа может завершиться досрочно
- Решение: разделение операций на блокирующиеся и неблокирующиеся

Оптимизация перебора в model checking

- Переход из задачи с номером x для узла i в задачу с номером y для узла j возможен, если $x < y \vee i = j$.
- Если переход сделать нельзя, но при этом остались необработанные задачи, то происходит подъём по дереву до тех пор, пока нельзя будет сделать переход, и после этого ветка удаляется
- Таким образом, сокращается перебор эквивалентных вариантов, и такие варианты не хранятся в памяти

Эффективность поиска ошибок

Алгоритм	**Ошибки / некорректные условия**
Exactly-once delivery	Не сохраняются id полученных сообщений
Алгоритм Лампорта	Нет FIFO доставки сообщений
Raft	Переменная **votedFor** не хранится персистентно
Raft	Может отказать половина узлов одновременно