Тестирование конкурентных структур данных на соответствие моделям согласованности

Морковкин Василий

2020

Аннотация

Чтение данных с записывающего устройства, проведение сетевых запросов и одновременная обработка поступающих команд - одни из задач, стоящих перед программным обеспечением. Результатом оптимизации затрат процессорного времени на решение этих задач стало появление ряда техник *конкурентного программирования*. Однако необходимость использования ресурсов, не поддерживающих одновременный доступ, зависимость от системного планировщика задач, а также особенности устройства процессорных кешей могут приводить к неопределенным или нежелаемым исполнениям компьютерных программ. Целью конкурентного программирования является предоставление механизмов и методик, которые бы позволили писать корректный, обладающий определенным поведением код. Одним из важнейших механизмов являются *структуры данных*, пригодные для использования в конкурентной среде. Для описания их свойств и предоставляемых гарантий был разработан ряд *моделей согласованности*.

Проверка алгоритмов на соответствие этим моделям может быть осуществлена с помощью методов формальной верификации. Однако для проверки конкретных их реализаций на языках программирования доступно лишь тестирование с перебором некоторого подмножества всех сценариев исполнения. В этой работе будут рассмотрены существующие модели согласованности конкурентных структур данных, разработаны варианты их тестирования, и, наконец, реализованы в виде библиотеки на языке программирования Scala.

Оглавление

Введение

1.1 Основные понятия

Введем основные понятия, необходимые в дальнейшем.

Операция — действие, состоящее из некоторого числа программных инструкций. Под этим словом будут подразумеваться отдельные взаимодействия со структурой данных (вставка, удаление, поиск и т.д.).

Процесс — последовательность операций, исполняемая любой единицей исполнения (системным процессом, системной нитью, нитью виртуальной машины, легковесной нитью и т.д.)

Последовательное исполнение — результат исполнения некоторых операций одним процессом.

Структура данных — формат хранения и управления данным, предоставляющий интерфейс для доступа к ним и их изменения.

Конкурентные операции — две или более операций, временные интервалы исполнения которых пересекаются.

Конкурентные процессы — процессы, исполняющие конкурентные операции.

Конкурентное исполнение — результат исполнения некоторых операций несколькими конкурентными процессами.

Конкурентная структура данных — структура данных, подчиняющаяся законам некоторой модели согласованности. Иными словами, структура данных,

пригодная для использования конкурентными процессами.

Модель согласованности — набор гарантий о предсказуемости результатов чтения, записи и изменения данных, позволяющий рассуждать о поведении компьютерной программы.

1.2 Цели работы

- Изучить проблемы, возникающие при конкурентном программировании,
- исследовать наиболее распространенные модели согласованности,
- разработать алгоритмы тестирования моделей согласованности,
- реализовать библиотеку для тестирования конкретных реализаций структур данных на выполнение выбранных моделей согласованности.

1.3 План работы

В *Главе 2* будут рассмотрены основные проблемы, а также причины их возникновения, с которыми сталкиваются разработчики при конкурентном программировании, такие как *состояние гонки*, *гонка на данных* и *взаимная блокировка*. Затем будут выделены модели согласованности конкурентных структур данных, такие как *последовательная согласованность*, *линеаризуемость* и различные ослабленные варианты линеаризуемости.

В *Главе 3* будет представлен анализ задачи тестирования конкурентных структур данных, а также предложен ряд конкретных алгоритмов тестирования. Также в главе пойдет рассуждение о хорошем дизайне для библиотеки с описанным функционалом тестирования.

Глава 4 включит в себя заметки о реализации библиотеки на языке программирования *Scala*. Выбор языка обусловлен его встроенными метапрограммированием, возможностью написания на нем удобных предметно-ориентированных языков, возможностью использования мощной стандартной библиотеки

java.util.concurrent, а также практической нуждой в подобной библиотеке. Все примеры кода в работе также будут приведены на языке *Scala* версии *2.13*.

Конкурентное программирование

2.1 Проблемы

В данном разделе мы рассмотрим ряд проблем, возникающих при помещении программ в конкурентную среду. Первая из них — гонка на данных.

2.1.1 Гонка на данных

Гонкой на данных называют [1] обращения к одному и тому же участку памяти, совершаемые из конкурентных процессов при условии, что хотя бы одно из обращений является обращением на запись, и обращения происходят не в рамках операций синхронизации. Под операциями синхронизации подразумевают, например, вызовы методов мониторов и барьеров памяти.

Рассмотрим пример, содержащий гонку на данных:

```
1  var flag: Boolean = false
2  def raiseFlag = { flag = true }
3
4  ForkJoinPool.commonPool.execute(DataRaceExample.raiseFlag _)
5  while (!flag) {}
```

Листинг 2.1: Пример гонки на данных

Результатом исполнения приведенного кода может стать зависание программы в бесконечном цикле. Причиной этого является копирование значений переменных, с которыми работает процесс, в кеш вычислительного процессора, и последующего чтения из него в то время, когда значение переменной уже было изменено. Данное копирование происходит с целью увеличения эффективности, так как доступ к процессорному кешу значительно быстрее [2] доступа к оперативной памяти.

Описанная проблема решается путем синхронизации обращений к переменных через примитивы синхронизации, встроенные в язык программирования (например, мониторов в Java). Для обнаружения гонок на данных были разработаны специальные инструменты [3, 4].

2.1.2 Состояние гонки

Другая проблема — *состояние гонки*. Несмотря на похожие названия, описанная ранее проблема не имеет прямого отношения к текущей. Гонка на данных возможна без состояния гонки и наоборот [5]

Состоянием гонки называют [6] ситуацию, в которой несколько процессов производят чтение и запись в разделяемую память, и при этом результат их операций зависит от порядка исполнения.

Рассмотрим пример кода, содержащий состояние гонки:

```
1  def transfer(amount: Int, from: AtomicInteger, to: AtomicInteger) = {
2    val fromBalance = from.get()
3    val toBalance   = to.get()
4    from.set(fromBalance - amount)
5    to.set(toBalance + amount)
6  }
7
8  transfer(10, 100, 100)
```

Листинг 2.2: Пример состояния гонки

Результат исполнения приведенного кода двумя процессами зависит от относительного порядка исполнения отдельных операций в этих процессах:

Таблица 2.1: Исполнение 1

Процесс 1	Процесс 2
from.get: 100	
to.get: 100	
from.set: 90	
	from.get: 90
	to.get: 100
	from.set: 80
to.set: 110	
	to.set: 110
Итог: from = 80, to = 110	

Таблица 2.2: Исполнение 2

Процесс 1	Процесс 2
from.get: 100	
to.get: 100	
	from.get: 100
	to.get: 100
	from.set: 90
from.set: 90	
to.set: 110	
	to.set: 120
Итог: from = 90, to = 120	

Видно, что в первом исполнении суммарное число денег на банковских счетах **from** и **to** уменьшилось, а во втором увеличилось, в то время как при исполнении на одном процессе оно не меняется. Данный пример показывает, что состояние гонки может появляться и в хорошо синхронизированном коде, в котором отсутствуют гонки на данных. Для рассуждения о программах, которые не подвержены таким проблемам, нам понадобится ввести понятия моделей согласованности, о которых речь пойдет далее.

2.1.3 Взаимная блокировка

Ещё одной частой причиной некорректного поведения конкурентных программ является состояние *взаимной блокировки*, при котором каждый процесс находится в ожидании совершения какого-то действия (отправки сообщения, отпускания блокировки) другим процессом. Система, находящаяся в таком состоянии, не делает прогресса.

Взаимная блокировка определяется одновременным выполнением следую-

щих условий [7]:

- Процессы требуют эксклюзивного владения ресурсами.
- Процессы, владеющие ресурсами, также ждут возможности завладеть некоторыми дополнительными ресурсами.
- Блокировка ресурса может быть отпущена только тем процессом, который им владеет в данный момент.
- В системе существует замкнутая цепь процессов, таких, что каждый процесс из цепи владеет одним или несколькими ресурсами, которые также необходимы следующему процессу в цепи.

Для наглядности рассмотрим пример:

```
1  val lock1 = new Object
2  val lock2 = new Object
3  Future {
4    lock1.synchronized {
5      lock2.synchronized()
6    }
7  }
8  Future {
9    lock2.synchronized {
10     lock1.synchronized()
11   }
12 }
```

Листинг 2.3: Пример взаимной блокировки

В зависимости от системного планировщика приведенная программа может как завершаться, так и повисать навечно в случае возникновения состояния взаимной блокировки, когда процесс первого блока *Future* не успевает захватить блокировку на объект *lock2* прежде процесса второго блока *Future*.

2.2 Модели согласованности

Все рассмотренные нами проблемы обладают общим свойством – они проявляются не всегда, а только при стечении определенных обстоятельств, не подконтрольных программисту, таких как состояние процессорных кешей и результат работы системного планировщика. Такое непостоянство, а также большое мест в коде, которые могут вызывать проблемы, делает сложным воспроизведение и исправление ошибок.

Поэтому была разработан формализм *моделей согласованности*, который вводит свойства корректности всех конкурентных исполнений некоторого кода, а также дает возможность рассуждать о композиции этих свойств при объединении корректных по отдельности участков кода.

В этом разделе мы рассмотрим несколько широко применяемых моделей согласованности, таких как *последовательная согласованность* и *линеаризумость*.

2.2.1 Последовательная согласованность

Говорят, что конкурентное исполнение является *последовательно согласованным* (англ. *sequentially consistent*), если существует последовательное исполнение с таким же результатом, в котором сохранялся бы относительный порядок операций каждого отдельного процесса. Участок кода называют последовательно согласованным, если в нем не возможны состояния взаимной блокировки, и все его конкурентные исполнения последовательно согласованы. [8]

Рассмотрим это понятие на примере. Пусть есть структура данных, которая поддерживает семантику “первым пришел – первым ушел” при исполнениях на одном процессе. Пусть она имеет следующий интерфейс:

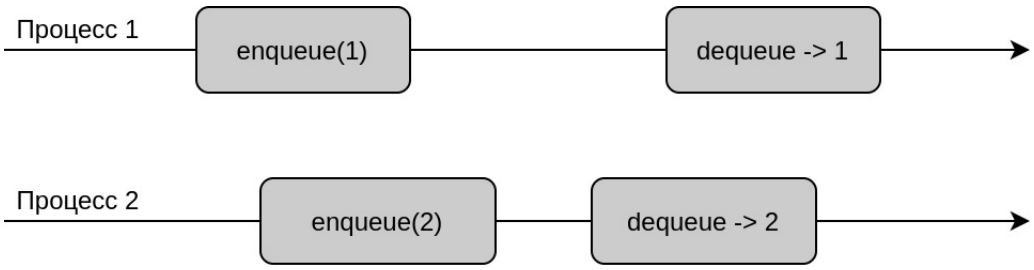
```
1  trait Queue {
2    def enqueue(e: Int): Unit
3    def dequeue: Int
```

```
4   }
```

Листинг 2.4: Интерфейс очереди

Если ее реализация утверждает, что она последовательно согласованна, то она не должна допускать подобных исполнений:

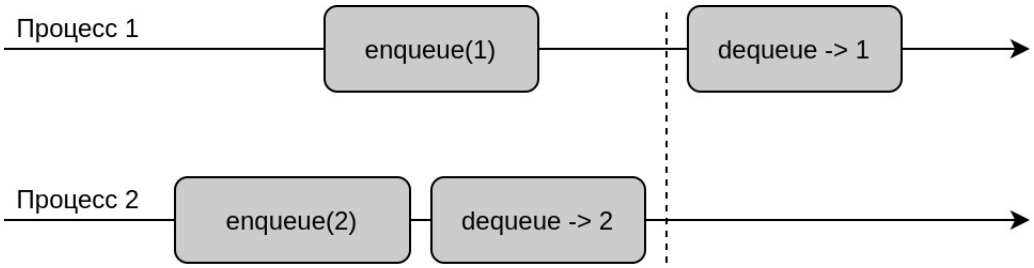
Рис. 2.1: Последовательно не согласованное исполнение



В данном случае нарушается требование существования исполнения на одном процессе, в котором операция *dequeue* вернула бы единицу дважды.

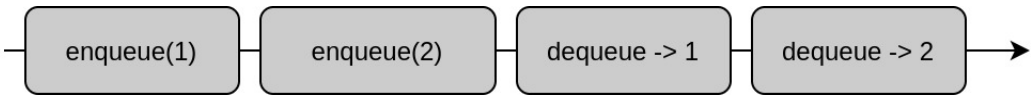
Однако подобные исполнения:

Рис. 2.2: Последовательно согласованное исполнение



допускаются, так как в данном случае все операции можно упорядочить в линейную историю, которая подчиняется спецификации объекта и сохраняет относительный порядок операций каждого из процессов:

Рис. 2.3: Объясняющее последовательное исполнение

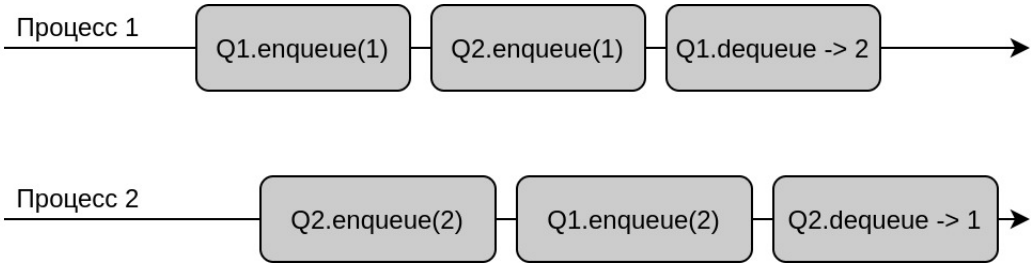


Корректность конкурентного исполнения, приведенного на рисунке 2.2, может показаться контр-интуитивной, поскольку операция *dequeue -> 2* заверши-

лась раньше, чем началась операция *dequeue -> 1*, в отличие от объясняющего последовательного исполнения на рисунке 2.3.

Также стоит отметить, что результат композиции последовательно согласованных объектов может не являться последовательно согласованным. Рассмотрим некоторую конкурентную историю для двух очередей *Q1* и *Q2*:

Рис. 2.4: Композиция последовательно согласованных объектов



Для каждой из очередей по отдельности можно составить объясняющие последовательные истории, однако для всего исполнения в данном случае этого сделать нельзя.

Более строгой моделью согласованности, которая учитывает временной порядок операций, а также обладает свойством композиции, является *линеаризуемость*.

2.2.2 Линеаризуемость

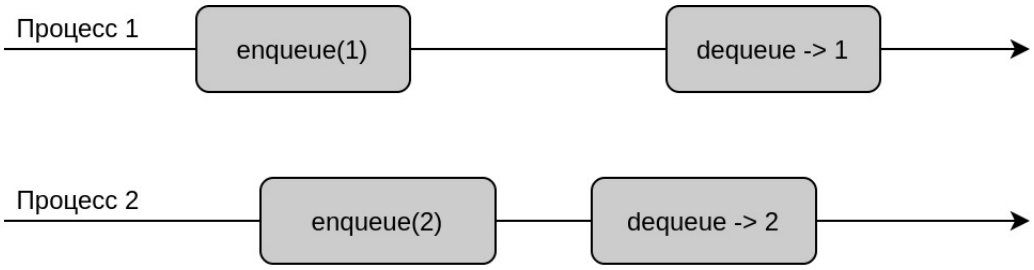
Конкурентное исполнение называется *линеаризуемым* (англ. *linearizable*), если существует корректное с точки зрения спецификации последовательное исполнение, в котором сохранен относительный порядок не пересекающихся во времени операций. Участок кода называют линеаризуемым, если в нем не возможны состояния взаимной блокировки, и все его конкурентные исполнения линеаризуемы. Поскольку операции, исполняемые одним и тем же процессом, не пересекаются во времени, линеаризуемость подразумевает последовательную согласованность и является более сильным свойством. [9]

Согласно определению, последовательно согласованное исполнение, изображенное на рисунке 2.2, не является линеаризуемым, поскольку не существует

последовательной истории, которая одновременно и удовлетворяла бы спецификации “первый пришел – первый ушел”, и сохраняла бы порядок непересекающихся во времени операций *dequeue -> 2* и *dequeue -> 1*.

Однако конкурентные операции по-прежнему могут быть упорядочены произвольным образом в объясняющем последовательном исполнении, поэтому такое исполнение линеаризуемо:

Рис. 2.5: Линеаризуемое исполнение



Его линеаризация приведена на рисунке 2.3

Линеаризуемость, в отличие от последовательной согласованности, является композирующимся свойством. [9] Это означает, что любое конкурентное исполнение кода, состоящего из операций над некоторыми объектами (например, структурами данных) линеаризуемо, если и только если каждый из этих объектов линеаризуем по отдельности. Это свойство очень полезно, так как позволяет выделять некоторые “строительные блоки”, облегчая рассуждения о поведении более сложных участков кода в конкурентной среде.

Данная работа фокусируется на рассмотрении последовательной согласованности и линеаризуемости, поскольку это наиболее часто используемые свойства. Однако стоит отметить, что существуют и применяются на практике и другие модели согласованности, такие как *статическая* (англ. *quiscent*) консистентность [10], *причинная* (англ. *causal*) консистентность [11] и *квази-линеаризуемость* (англ. *quasi-linearizability*) [12].

Проверка моделей согласованности

В предыдущей главе мы рассмотрели некоторые удобные свойства, которые помогают рассуждать о поведении кода в многопроцессорных системах. Однако утверждения о том, что конкретные реализации структур данных обладают этими свойствами, нуждаются в проверке. Здесь можно выделить два основных подхода:

- *Формальная проверка* (англ. *verification*). Данный подход заключается в предоставлении доказательства корректности конкретного кода при фиксированной модели памяти. Это самый надежный подход из существующих, однако эта проблема решена лишь для некоторых классов задач и требует значительных усилий [13]. Например, решения для проверки на линеаризуемость основаны на поиске точек линеаризации [14]. Однако в некоторых структурах данных точек линеаризации не существует [9]. Статический анализ кода, отсутствие универсальности и требование знаний о моделях памяти, теории моделей согласованности и методов верификации делает затруднительным широкое применение этого подхода.

- *Тестирование* (англ. *model checking*). Представляет из себя перебор возможных исполнений программы и последующую проверку результатов исполнения относительно последовательной спецификации. Это не может га-

рантировать корректность проверяемого кода, однако не требует глубокого анализа и хорошо работает на практике, находя большое количество ошибок. [15]

В данной главе мы сосредоточимся на алгоритмах тестирования конкурентных структур данных на соответствие моделям последовательной согласованности и линеаризуемости, а также разработаем набор требований для библиотеки, реализующей данный функционал.

3.1 Постановка задачи

Пусть есть некоторая реализация конкурентной структуры данных. Обозначим количество доступных в ней операций через M. Пусть каждый из T конкурентных потоков совершает по L операций. Тогда количество возможных конкурентных сценариев исполнения без учета симметрии равно $(C_M^L)^T$, где C_M^L – количество способов составить сценарий исполнения для одного потока.

Кроме того, результаты запусков каждого из сценариев будут различны из-за отсутствия каких-либо гарантий от планировщика задач. При этом для каждого запуска некоторого сценария еще необходимо проверить, удовлетворяет ли он заявленной модели согласованности. Как для последовательной согласованности, так и для линеаризуемости эта задача является NP-полной [16].

Стремительный рост пространства состояний, а также неподконтрольность системного планировщика делает невозможным проверку абсолютно всех возможных ситуаций. Однако на практике это и не требуется. Опыт показывает, что для обнаружения всех известных ошибок обычно достаточно проводить тестирования на двух-трех конкурентных процессах со сценариями, не превосходящими по длине пяти-десяти операций [15].

3.2 Алгоритмы тестирования

Для проверки последовательной согласованности и, в особенности, линеаризуемости было предложено множество алгоритмов [17]. В данном разделе мы остановимся на простейших из них.

У алгоритмов проверки обоих моделей можно выделить общую часть, которая будет различаться лишь реализацией функции *possibleOperations*:

```
1  def isValid(‘history‘: [[operation]], ‘state‘: impl): bool = {
2    if ‘history‘ is empty { return true }
3    else {
4      for operation ‘op‘ in possibleOperations(‘history‘) {
5        let ‘res‘ be the result of ‘op‘ in ‘history‘
6        run ‘op‘ on ‘state‘
7        if ‘op‘ results in ‘res‘ &&
8            isValid(‘history‘ - ‘op‘, ‘state‘) { return true }
9        else {
10         undo ‘op‘ on ‘state‘
11       }
12     }
13     return false
14   }
15 }
```

Листинг 3.1: Общий алгоритм проверки

Прокомментируем алгоритм построчно:

- *Строка 1* – на вход алгоритм принимает *history* - историю конкурентного исполнения, включающая информацию о всех вызванных операциях и их результатах, а также *state* - реализацию тестируемой структуры данных с возможностью “отмены” операций (о способах реализации отмены мы поговорим позднее).
- *Строка 2* – пустая история всегда корректна.
- *Строки 3-4* – если история не пуста, то начать перебор всех допустимых

операций конкурентного исполнения, которые могли бы стать следующими в объясняющей последовательной истории. Вычисление этого списка кандидатов в методе *possibleOperatoins* зависит от проверяемой модели согласованности.

- *Строка 5* – обозначаем результат исполнения рассматриваемой операции-кандидата за *res*.
- *Строки 6-11* – применяем рассматриваемую операцию к *state*. Если результат исполнения операции совпал с *res*, продолжаем проверку корректности для оставшейся истории. Если проверка для оставшейся истории успешна, то сообщаем, что *history* корректна с точки зрения выбранной модели согласованности.
- *Строка 13* – сообщаем, что *history* не является корректной, поскольку не нашлось объясняющих ее последовательных исполнений.

3.2.1 Последовательная согласованность

В случае последовательной согласованности функция *possibleOperations* реализуется как список следующих по очереди операций каждого из процессов в программном порядке:

```
1 def possibleOperations(‘history‘: [[operation]]): [operation] = {
2   let ‘res‘ be an empty list []
3   for ‘processHistory‘ in ‘history‘ {
4     if ‘processHistory‘ is not empty {
5       add first element of ‘processHistory‘ to ‘res‘
6     }
7   }
8   return ‘res‘
9 }
```

Листинг 3.2: Поиск операций-кандидатов для последовательной согласованности

3.2.2 Линеаризуемость

Для линеаризуемости помимо программного порядка необходимо учитывать также и временной порядок операций, поэтому реализация функции *possibleOperations* отличается:

```
def possibleOperations(‘history‘: [[operation]]): [operation] = {
  let ‘res‘ be an empty list []
  for ‘processHistory‘ in ‘history‘ {
    if ‘processHistory‘ is not empty {
      add first element of ‘processHistory‘ to ‘res‘
    }
  }
  sort ‘res‘ by the end time of each operation
  return ‘res‘
}
```

Листинг 3.3: Поиск операций-кандидатов для линеаризуемости

3.3 Разработка требований к библиотеке

3.3.1 Интерфейс

3.3.2 Конфигурация

Реализация библиотеки

4.1 Интерфейс

4.2 Генерация тестовых сценариев

4.3 Запуск тестовых сценариев

4.4 Валидация результатов исполнения

4.5 Генерация отчета тестирования

Выводы

Литература

[1] J. Erikson et al. Effective data-race detection for the kernel. *Microsoft Research*, 2010.

[2] J. Dean. Latency numbers every programmer should know.

[3] K. Serebryany. Threadsanitizer – data race detection in practice. 2009.

[4] Jep: Java thread sanitizer.

[5] J. Regehr. Race condition vs. data race, 2011.

[6] S. Carr et al. Race conditions: A case study. 2001.

[7] E. Coffman et al. System deadlocks. 1971.

[8] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. 1979.

[9] M. Herlihy. Linearizability: a correctness condition for concurrent objects. 1990.

[10] N. Shavit. Counting networks. *Journal of the ACM 41(5)*, 1994.

[11] M. Ahamad et al. Causal memory: Definitions, implementation and programming. *Technical Report GIT-CC-93/55*, 1994.

[12] A. Yehuda et al. Relaxed consistency for improved concurrency. *Principles of distributed systems. Lecture notes in Computer Science*, 2010.

[13] E. Condon. Automatable verification of sequential consistency. *Theory of Computing Systems, 36(5)*, 2003.

[14] V. Vafeiadis. Automatically proving linearizability. *Computer Aided Verification. Lecture Notes in Computer Science*, 2010.

[15] S. Burckhardt. Line-up: a complete and automatic linearizability checker. *Proceedings of the 31st International Conference on International Conference on Parallel Computing*, 2010.

[16] P. Gibbons et al. The complexity of sequential consistency. *Proceeding of the Fourth IEEE Symposium on Parallel and Distributed Processing*, 1992.

[17] G. Lowe. Testing for linearizability. *Concurrency and Computation. Practice and experience*, 2016.