

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ	4
ВВЕДЕНИЕ	6
ГЛАВА 1. АЛГОРИТМ В БАЗЕ ДАННЫХ ВК	9
1.1. VIEWSTAMPED REPLICATION	9
1.1.1. Хранимое состояние на репликах	9
1.1.2. Normal Operation Protocol	9
1.1.3. View Change Protocol	10
1.2. ОПТИМИЗАЦИИ В БАЗЕ ДАННЫХ ВК	11
1.2.1. View блоки в логе	11
1.2.2. Download Protocol	12
1.3. ПОСТАНОВКА ЗАДАЧИ	13
ВЫВОДЫ ПО ГЛАВЕ 1	13
ГЛАВА 2. TLA+ СПЕЦИФИКАЦИИ	14
2.1. РАЗЛИЧИЯ СПЕЦИФИКАЦИЙ	14
2.2. АБСТРАКТНАЯ СПЕЦИФИКАЦИЯ	14
2.2.1. Переменные и константы абстрактной спецификации	14
2.2.2. Типовой инвариант и предположения абстрактной модели	16
2.2.3. Переходы абстрактной спецификации	18
2.2.4. Инварианты абстрактной спецификации	22
2.2.5. Fairness и Liveness абстрактной спецификации	22
2.3. ДЕТАЛЬНАЯ СПЕЦИФИКАЦИЯ	24
2.3.1. Переменные и константы детальной спецификации	25
2.3.2. Типы в детальной спецификации	25
2.3.3. Действия в детальной модели	27
2.3.4. Имплементация абстрактной спецификации	30

Выводы по главе 2	31
Глава 3. TLC модели	32
3.1. TLC модель абстрактной спецификации	32
3.1.1. Создание абстрактной модели	33
3.1.2. Запуск абстрактной модели	35
3.2. TLC модель детализированной спецификации	37
3.2.1. Создание детализированной модели	38
3.2.2. Запуск детализированной модели	40
3.3. Результаты	42
Выводы по главе 3	45
Заключение	46
Библиографический список	47

ВВЕДЕНИЕ

Во внутренней инфраструктуре VK используются движки для хранения информации [1–2]. Это различные самописные базы данных, созданные для оптимизации хранения разных видов данных.

Движки расположены на разных виртуальных или железных машинах. У них могут случаться отказы. Необходимо поддерживать сохранность и консистентность данных в случае разных сбоев в работе машин и сети, иначе, например, пользователи потеряют свои сообщения или купленные предметы в социальной сети.

Всё это является типичной задачи репликации лога и решается алгоритмами распределённого консенсуса: Paxos [3], Raft [4], Viewstamped Replication [5]. Последний вариант был взят за основу собственного алгоритма в VK.

Для автоматической проверки корректности написанного кода применяются различные методы - тесты, фазеры. Но они не дают формального доказательства и не позволяют быть полностью уверенными в правильности реализованного алгоритма.

Существуют методы и инструменты формальной верификации распределённых систем и параллельных алгоритмов. Одним из самых популярных является язык TLA+ [6] и инструмент TLC [7] для него. На языке TLA+ можно высокоуровнево описывать алгоритм или систему — составление TLA+ спецификации. С помощью TLC можно строить модель по спецификации и проверять выполнение требуемых свойств, которыми должна обладать система. Такая техника более отдалена от кода, чем тесты и фазеры, однако она автоматически проверяет все возможные исполнения — не только случайно сгенерированные фазером или описанные в сценариях тестов.

В спецификации на языке TLA+ описывается некая абстрактная модель мира, в котором работает система. Делается это через переменные, описывающих, например, мета информацию на репликах, журнал лога на

каждой из них или множество посланных сообщений в сети. Множество значений всех переменных в один момент времени — это одно состояние. В спецификации перечисляются возможные события, меняющие их значение, например, получение сообщения на реплике, приём запроса от клиента или падение машины.

Инструмент TLC model checker строит по этим переменным и событиям ориентированный граф, в котором вершинам соответствуют состояния, а рёбрам между ними — возможные переходы, описанные в спецификации на TLA+. Для каждой вершины проверяются инварианты модели — например, консистентность закомиченных элементов лога или единственность лидера в одних выборах.

Существуют сложности, связанные с этими моделями. Если мы возьмём большую детализацию для спецификации, граф состояний может быть очень большой и проверки на нём слишком долгими. Это бывает, если описывать в модели много деталей реализации, лишних переменных, из-за которых граф будет экспоненциально разрастаться.

С другой стороны спецификация на TLA+ может быть слишком абстрактной, из-за чего инженерам может быть сложно убедиться, что она соответствует реальному коду или алгоритму. Это бывает, если, наоборот, упрощать модель и упускать детали. Получается, что нужно следить за тем, чтобы было приближено к реальному коду, но в тоже время и не сильно детализировано.

Для этого Лесли Лампорт добавил в TLA+ возможность имплементирования спецификаций. Одна модель может реализовывать другую, если во время исполнения одной корректно выполняются шаги и в другой. Поскольку это является логической импликацией, то по транзитивности первая спецификация наследует свойства второй и их можно отдельно не проверять.

Спецификации, предлагаемые в этой работе, проверены на корректность на некоторых ограниченных пространствах состояний. Кроме того,

детализированная спецификация корректно реализует абстрактную. Благодаря этому модель можно изучать, начиная с абстрактной, постепенно переходя на более детализированную.

ГЛАВА 1. АЛГОРИТМ КОНСЕНСУСА В БАЗЕ ДАННЫХ ВК

Вначале рассмотрим алгоритм, который предстоит верифицировать.

1.1. VIEWSTAMPED REPLICATION

За основу алгоритма разработчиками была взята статья про Viewstamped Replication [5]. Авторы статьи предлагают отказоустойчивый алгоритм в том смысле, что он будет корректно работать, даже когда упадут не более f машин, если всего есть $2 \cdot f + 1$ реплик.

1.1.1. Хранимое состояние на репликах

На репликах хранится топология кластера (упорядоченный список всех нод), номер данной реплики, текущий номер view (номер последнего голосования), текущий статус реплики (Normal, ViewChange или Recovering), журнал лога и commit number. Это основные параметры для нашей спецификации. Порядок реплик в списке важен и должен быть одинаков на всех - именно в этом порядке выбираются лидеры в голосованиях. Например, в случае 3 реплик, первая будет лидером во всех view с номерами 0, 3, 6, вторая — в 1, 4, 7 и так далее. Это даёт преимущество над алгоритмами с выбором лидера по числу голосов, поскольку, во-первых, в нашем случае лидер каждый раз определён и не приходится отправлять сообщения с голосами, а во-вторых, при выборах может никто и не набрать кворум голосов. Однако, в predetermined порядке лидеров есть и минус, если новый мастер изолирован по сети от кластера или если он упал — в таком случае потратится время на начало ещё одних выборов, хотя эта машина могла не участвовать в голосовании. Журнал лога должен храниться в энергонезависимой памяти, чтобы не теряться в случае отказа узла.

1.1.2. Normal Operation Protocol

Когда реплика, считающая себя лидером, получает запрос от клиента, она добавляет его в конец лога и рассылает всем репликам сообщение типа Prepare. Нода, получившая актуальный Prepare, тоже добавляет запрос в лог и

отсылает лидеру в ответ PrepareOk. Актуальный для неё тот, у которого такой же номер view, как и у неё самой, а так же OpNumber которого (позиция в логе) на 1 больше длины журнала лога машины.

Как только лидер набирает кворум ответов ($f + 1$), включая себя, он считает операцию законченной и увеличивает commit number. Поскольку в данный момент более, чем на половине реплик есть клиентский запрос, то он останется в оперативной памяти хотя бы одной реплики (если f машин упадут). Упавшие машины будут восстанавливать потерянную информацию, а у тех, кто остался, будет приоритетная цепочка в протоколе, который обсудим следующим.

Так же мастер периодически отсылает репликам Commit сообщение для того, чтобы все могли коммитить и исполнить операцию, а так же, чтобы ноды узнавали, что мастер жив, и не запускали новое голосование.

1.1.3.View Change Protocol

Если же у кого из реплик сработал таймер на голосование, то она переходит в статус ViewChange, увеличивает view и рассылает всем сообщение StartViewChange. Другая реплика может получить это сообщение и сделать то же самое. Итого все ноды, переходящие в фазу смены лидера, отсылают всем StartViewChange. Делают они это по собственному таймеру или сообщению от другой реплики с номером view, большим, чем у неё.

Как только нода набирает f сообщений StartViewChange от других, то это означает, что минимум кворум реплик перешёл в новый view, и старый лидер точно не будет коммитить новые запросы, так как реплика отвечает только на Prepare с её номером view. Сразу после достижения нужного количества сообщений она отсылает primary реплике в этом view сообщение DoViewChange, в котором записывает свою информацию: её полный лог, commit number, last normal view (Максимальный номер view, в котором нода была в статусе Normal). Это нужно для того, что будущий мастер выбрал лучшую цепочку и скачал её себе.

Далее, как только будущий лидер (но ещё не подтверждённый) получает *f* DoViewChange сообщений без учёта себя, то считает, что кворум реплик знает про новые выборы. Он выбирает реплику, лог которой возьмёт себе и рассылает другим. Вначале выбирается реплика с максимальным last normal view. Дальше, если таких несколько, то из них берётся с большей длиной лога. Мастер копирует полностью этот лог себе. Потом отсылает всем сообщение StartView со скопированным логом и commit number.

Когда нода в статусе ViewChange получает StartView с таким же номером view или же в любом статусе сообщение с view выше своего — то ставит свой view number равный view из сообщения и статус Normal. Теперь реплика отвечает новому мастеру PrepareOk сообщениями на старые незакоммиченные блоки, чтобы тот мог закомитить то, что не успели старые лидеры. А после этого нода работает по обычному Normal протоколу.

1.2. ОПТИМИЗАЦИИ В БАЗЕ ДАННЫХ ВК

Разработчики движков в VK реализовали алгоритм из статьи про Viewstamped Replication и сделали некоторые изменения и собственные оптимизации.

1.2.1. View блоки в логе

Были добавлены специальные View мета блоки в журнал лога. Каждый такой элемент в истории операций свидетельствует о том, что в данный момент реплика завершила View Change протокол, перешла к новому view и получила Normal статус. View блок с текущим view number коммитит лидер сразу после установки себе лога из лучшей цепочки и прямо перед отправкой StartView сообщений. На этот блок другие реплики так же отвечают PrepareOk, как и на обычные, то есть он реплицируется по обычным правилам, как и клиентские запросы. В этом есть некая унификация — мы реплицируем начало нового view так же, как команды к движкам. Если на кворум реплик этот блок не попал перед началом новой смены view, то может быть такое, что новый лидер не

увидит его среди цепочек в сообщениях DoViewChange — то же самое возможно с обычным запросом клиента, который лидер не успел реплицировать на более половины машин. А если кворум успел скачать этот блок — то в будущем он не потеряется точно так же, как и команда к движку.

Помимо этого эти блоки используются в качестве last normal view, нужного в DoViewChange сообщениях — это значение ставится равному view последнего такого блока в журнале.

1.2.2. Download Protocol

Передача всего лога в DoViewChange и StartView сообщениях на практике, естественно, не происходит. Вместо этого разработан собственный протокол скачивания, Download Protocol. Реплика, которая хочет скачать лог до какого-то момента с другой посылает той сообщение StartDownload с некоторой информацией об имеющемся у себя журнале. Другая машина сравнивает эту информацию со своей, чтобы определить, есть ли расхождения в журналах — несовпадающие элементы на одинаковых позициях. Потом начинает отсылать обратно сообщения DownloadChunk, в которых передаёт куски лога. Первая нода добавляет себе недостающие элементы журнала, либо заменяет несовпадающие. Это происходит и после того, как новоизбранный мастер выбрал реплику, цепочку которой возьмёт себе в новый view, и после получения нодой сообщения StartView.

Если у реплики, которая в данный момент скачивает что-то с другой, сработает таймер на начало нового view, то скачивание остановится — оно продолжится в будущем с другой реплики по протоколу View Change.

Так же DownloadChunk сообщения используются в Normal Operation протоколе вместо Prepare — при добавлении нового клиентского запроса мастер бродкастит (рассылает всем репликам) DownloadChunk с новым элементом лога. Это унифицирует то, как одни реплики копируют журнал с других, так как это всегда происходит через сообщения одного вида.

1.3. ПОСТАНОВКА ЗАДАЧИ

Необходимо верифицировать описанный алгоритм на языке TLA+ [6]. Но из-за сложностей TLA+ спецификаций одной модели может быть мало — она будет слишком абстрактная или, наоборот, слишком подробная и большая, из-за чего на ней тяжело будет проверять свойства. Поэтому хочется написать две спецификации и проверить, что первая удовлетворяет свойствам, а вторая больше приближена к коду и удовлетворяет первой. В следующей главе будет описано подробнее, чем будут различаться модели, и почему подробная приближена к коду.

Цель:

- Верифицировать алгоритм репликации движков моделями с разными уровнями абстракции

Задачи:

- Написать абстрактную TLA+ спецификацию
- Проверить выполнение свойств на абстрактной TLC модели
- Написать детализированную TLA+ спецификацию, реализующую первую
- Проверить выполнение абстрактной TLC модели в детальной

ВЫВОДЫ ПО ГЛАВЕ 1

В этой главе описан алгоритм из статьи про Viewstamped Replication и его реализация в сервисе репликации движков в ВК, а также сформирована цель работы и задачи для её достижения.

ГЛАВА 2. TLA+ СПЕЦИФИКАЦИИ

Для проверки корректности алгоритма репликации движков необходимо написать спецификацию на TLA+. В введении мы рассказали про проблемы у спецификаций и соответствующих им TLC моделей. Во-первых, описание системы на языке TLA+ может быть слишком абстрактным и не похожим на реальный код. Во-вторых граф состояний в TLC модели может быть слишком большим из-за, наоборот, очень подробной модели. Поэтому хочется написать две спецификации: 1) более детальную, для большего соответствия коду, и 2) абстрактную, для проверки свойств.

2.1. РАЗЛИЧИЯ СПЕЦИФИКАЦИЙ

Хочется, чтобы абстрактная модель задавала то, что происходит в системе. Например, мастер добавляет новый запрос пользователя, реплика копирует с мастера этот запрос или нода становится лидером, выбирая лучшую цепочку. А детальная спецификация должна показывать то, как это происходит. Одна декларативно описывает "что", вторая описывает "как".

Основная идея состоит в том, чтобы в абстрактной спецификации не было сообщений между репликами, а в детальной они были. Тогда в первой переменные с состояниями машин будут обмениваться значениями друг с другом напрямую — реплики копировать лог мастера, например — а вторая будет отвечать на вопрос, как они это делают — через сообщения.

2.2. АБСТРАКТНАЯ СПЕЦИФИКАЦИЯ

Вначале опишем абстрактную спецификацию.

2.2.1. Переменные и константы абстрактной спецификации

В TLA+ есть константы и переменные. Константы задаются для модели извне и не меняются для всего графа состояний. А значения переменных, наоборот, позволяют различить вершины графа. Они меняются в переходах, разрешённых в модели. Переменные и константы абстрактной модели представлены на Листинге 1.

Листинг 1 — константы и переменные абстрактной спецификации

```

CONSTANTS Replica, Quorum

\* Replica Status
CONSTANTS Normal, ViewChange, Recovering

\* Client operation
CONSTANT Operation

\* types of log blocks
CONSTANTS RequestBlock, ViewBlock

\* Special value
CONSTANT None

\* Sequence with all replicas (for view selection)
CONSTANT ReplicaSequence

\* For state space limitation
CONSTANT MaxRequests, MaxViews

\* State on each replica
VARIABLE replicaState

vars == <<replicaState>>

```

- *Replica* — это множество модельных значений, соответствующих репликам. Например, если $Replica = \{r1, r2, r3\}$, значит, у нас есть 3 реплики, к которым можно обращаться, как $r1, r2, r3$.
- *Quorum* — множество всех кворумов, то есть подмножеств реплик таких, что каждый с каждым пересекается хотя бы по одной. Обычно для трёх реплик это $\{\{r1, r2\}, \{r2, r3\}, \{r1, r3\}\}$.
- *Normal, ViewChange, Recovering* — константы для обозначения статусов реплики.
- *Operation* — это множество операций к движкам, которые мы и реплицируем на машинах. Можно абстрактно обозначить, как $\{o1, o2\}$.

Именно два значения, потому что когда мы будем проверять согласованность журналов на разных машинах, мы сможем различить одинаковые операции хранятся на одной позиции или нет. Для этого хватает и двух значений. Если существуют некорректные исполнения с тремя разными запросами, в которых нарушается консистентность логов, то существует исполнение и с двумя, в котором это нарушение обнаружится.

- `RequestBlock`, `ViewBlock` — константы для различения в журнале запросов к движкам и `View` блоков.
- `None` это специальное значение, не равное ничему другому.
- `ReplicaSequence` это порядок реплик для выборов. Например, $[r1, r2, r3]$.
- `MaxRequests`, `MaxViews` — это числовые константы, задающие ограничения для рассматриваемого пространства состояний.
- `replicaState` — единственная переменная — обозначает состояния всех нод. Её возможные значения будут специфицированы далее.

2.2.2. Типовой инвариант и предположения абстрактной модели

В TLA+ принято обозначать инвариант с допустимыми типами для переменных как `TypeOK`. Для данной спецификации приведён типовый инвариант на Листинге 2.

`Statuses` задаёт множество возможных статусов реплики. Все значения — являются константами модели.

`LogEntry` обозначает множество возможных элементов лога. Это либо структура с полями `type`, `opNumber` и `op`, либо с полями `type` и `view`. В первом случае `type` принимает значение из множества $\{RequestBlock\}$ (то есть всегда равен `RequestBlock`), `opNumber` из множества `Nat` (натуральные числа и 0), `op` из множества `Operation` (мы его задавали, как константа модели). Во втором случае `type` всегда `ViewBlock`, а `view` — число из `Nat`.

В `TypeOK` записано утверждение, что `replicaState` — это функция из реплики в структуру с полями `viewNumber`, `status`, `log`, `downloadReplica`,

commitNumber; их множества допустимых значений указаны рядом с ними. Стоит отметить, что $Seq(LogEntry)$ обозначает последовательность из LogEntry.

Листинг 2 — типы и предположения

```

Statuses == {Normal, ViewChange, Recovering}

LogEntry == [type: {RequestBlock}, opNumber: Nat, op: Operation]
           \cup [type: {ViewBlock}, view: Nat]

TypeOK == \wedge replicaState \in [
  Replica -> [
    viewNumber: Nat,
    status: Statuses,
    log: Seq(LogEntry),
    downloadReplica: Replica \cup {None},
    commitNumber: Nat
  ]
]

ASSUME QuorumAssumption == \wedge \A Q \in Quorum : Q \subseteqq Replica
                          \wedge \A Q1, Q2 \in Quorum : Q1 \cap Q2 \# {}

ASSUME IsFiniteSet(Replica)

```

Когда мы будем запускать модель, то будем ставить TypeOK инвариантом — то есть утверждением, которое должно быть истинным во всех допустимых состояниях модели. TLC model checker будет проверять это во всех вершинах графа состояний.

QuorumAssumption утверждает, что все элементы из Quorum должны быть подмножеством реплик, и что все они должны попарно пересекаться ($\wedge A$ обозначает квантор всеобщности). Это и есть определение кворума. Ключевое слово ASSUME означает предположение, которое должно выполняться для модели. TLC проверит его и сообщит, если оно нарушается.

Утверждение $IsFiniteSet(Replica)$ говорит, что $Replica$ — не бесконечное множество; если оно нарушается, то TLC будет бесконечно пробовать выполнить почти любое действие и проверка будет бессмысленна.

2.2.3. Переходы абстрактной спецификации

Листинг 3 — Начальное состояние и пример перехода

```

Init ==  $\wedge$  replicaState = [r \in Replica |-> [
    viewNumber |-> 0,
    status |-> Normal,
    log |-> << [type |-> ViewBlock, view |-> 0] >>,
    downloadReplica |-> None,
    commitNumber |-> 0
  ]
]

AddClientRequest(r, m) ==
   $\wedge$  replicaState' = [replicaState EXCEPT ![r].log = Append(@, m)]

RecieveClientRequest(p, op) ==
   $\wedge$  IsPrimary(p)
   $\wedge$  Status(p) = Normal
   $\wedge$  ~IsDownloading(p)
   $\wedge$  AddClientRequest(p, [type |-> RequestBlock,
    opNumber |-> OpNumber(p) + 1,
    op |-> op])

```

На Листинге 3 приведён $Init$ — традиционное обозначение начального состояния модели — и $RecieveClientRequest$ — пример возможного перехода. В $Init$ записано утверждение про значение полей $replicaState$ для всех машин — $viewNumber$ равно нулю, статус $Normal$, в логе одна запись про начало $view$ с номером 0, $downloadReplica$ (реплика, с которой мы скачиваем) никакая, $commitNumber$ равно нулю.

$RecieveClientRequest$ показывает, как обычно выглядят переходы в модели. Это конъюнкция утверждений двух типов: предикатов действия и новых значений переменных. Конъюнкты $IsPrimary(p)$, $Status(p) = Normal$

и $\sim IsDownloading(p)$ — предикаты. Действие может выполняться в каком-то состоянии мира, если они все равны True. Первое обозначает, что реплика r сейчас лидер своего view, второе говорит, что её статус Normal, третье — r ничего не скачивает (её `downloadReplica` равно None). `AddClientRequest` можно назвать макросом, вместо которого подставится его определение, представленное на том же Листинге 3. В нём один конъюнкт, который уже является утверждением про новое значение переменной `replicaState`. В этом специфическом синтаксисе TLA+ записано, что новое значение списка `replicaState[r].log` станет равно старому, к которому добавили (`Append(@, m)`) структуру m .

Само значение структуры m — $[type \mid -> RequestBlock, opNumber \mid -> OpNumber(p) + 1, op \mid -> op]$. Остаётся вопрос, откуда берётся значение реплики r и операция op в параметрах перехода `RecieveClientRequest`. Это будет понятно далее.

Листинг 4 — Все допустимые действия в Next

```

NormalOperationProtocol ==
  ∨ ∃ r ∉ Replica, op ∉ Operation: RecieveClientRequest(r, op)
  ∨ ∃ r ∉ Replica: RecievePrepare(r)
  ∨ ∃ p ∉ Replica: AchievePrepareOkFromQuorum(p)
  ∨ ∃ r ∉ Replica: RecieveCommit(r)

ViewChangeProtocol ==
  ∨ ∃ r ∉ Replica: TimeoutStartViewChanging(r)
  ∨ ∃ r ∉ Replica: RecieveStartViewChange(r)
  ∨ ∃ r ∉ Replica: AchieveDoViewChangeFromQuorum(r)
  ∨ ∃ p ∉ Replica: MasterDownloadBeforeView(p)
  ∨ ∃ r ∉ Replica: RecieveStartView(r)
  ∨ ∃ r ∉ Replica: ReplicaDownloadBeforeView(r)

Next == ∨ NormalOperationProtocol
        ∨ ViewChangeProtocol
        ∨ Finishing

```

На Листинге 4 дано определение Next — традиционное для TLA+ обозначение дизъюнкции всех возможных действий, разрешённых в спецификации. Макросы NormalOperationProtocol и ViewChangeProtocol группируют для удобства действия из разных протоколов, описанных в Главе 1.

Здесь видно, откуда берётся реплика и операция для RecieveClientRequest — из множеств всех реплик и операций, соответственно. Обозначение $\exists E$ — квантор существования.

Действия в NormalOperationProtocol:

- RecieveClientRequest — получение репликой, которая считает себя лидером (таких может быть одновременно несколько в разных view), запроса от клиента. Реализована, как мы видели просто выбором случайной операции из всех возможных. Словно мастер сам генерирует следующий запрос движку.
- RecievePrepare — получение репликой клиентского запроса от мастера.
- AchievePrepareOkFromQuorum — мастер убедился, что на кворум реплик реплицирован запрос и закоммитил его.
- RecieveCommit — нода получила уведомление о коммите от мастера.

Теперь действия из ViewChangeProtocol:

- TimeoutStartViewChanging — реплика решила сменить лидера и начать новый view.
- RecieveStartViewChange — нода узнала от другой о новой смене view.
- AchieveDoViewChangeFromQuorum — новый лидер достиг кворума DoViewChange сообщений. Причём в этой абстрактной спецификации сами сообщения не моделируются. Описывается лишь то, что получение этих сообщений могло произойти, если половина других реплик находится в таком же view или выше. А в детальной спецификации будет моделироваться получение сообщений и при запуске модели проверяться, что такие манипуляции с сообщениями подходят под высокоуровневую спецификацию.

- `RecieveStartView` — реплика узнала от лидера о начале нового view.
- `MasterDownloadBeforeView` — скачивание мастером элемента лога с реплики с лучшей цепочкой.
- `ReplicaDownloadBeforeView` — скачивание нодой элемента лога с нового лидера.
- `Finishing` — это специальное действие, добавленное, чтобы понимать, реплики пришли к итоговому консенсусу. Его определение дано на Листинге 5.

Листинг 5 — определение `Finishing`

```

Finishing ==
  ^ LET r == ReplicaSequence[1]
    \* All Committed
  IN ^ CommitNumber(r) = OpNumber(r)
    \* MaxRequests commands are stored
    ^ RequestBlockCount(Log(r)) = MaxRequests
    \* All replicas equal
    ^ \A r1 \in Replica:
      ^ Log(r1) = Log(r)
      ^ CommitNumber(r1) = CommitNumber(r)
      ^ ViewNumber(r1) = ViewNumber(r)
      ^ Status(r1) = Normal
      ^ DownloadReplica(r1) = None
    ^ UNCHANGED <<replicaState>>

```

В нём утверждается, что у первой реплики закоммичен весь её лог, количество клиентских запросов к движкам в нём максимально возможно, и что на всех остальных машинах полностью совпадает с ней журнал, позиция коммита, номер View, статус Normal и никто ничего не скачивает.

Это действие — единственное, которое никак не меняет состояние реплик, поэтому здесь видим конструкцию `UNCHANGED << replicaState >>`, которая об этом и говорит. Позже мы поймём полностью, как этот переход нам полезен.

2.2.4. Инварианты абстрактной спецификации

Помимо определения состояния системы и разрешённых переходов в ней необходимо описать что-то, что позволит проверять её корректность. Мы уже определяли один инвариант — `TypeOK`, гарантирующий правильность значения переменной `replicaState`

Листинг 6 — Инварианты абстрактной модели

```
CommittedLogsPreficesAreEqual ==
  \A r1, r2 \in Replica:
    \A i \in DOMAIN Log(r1)
      \cap DOMAIN Log(r2)
      \cap 1 .. Min({CommitNumber(r1),
                    CommitNumber(r2)}):
        Log(r1)[i] = Log(r2)[i]

KeepMaxRequests == \A r \in Replica: RequestBlockCount(Log(r)) <=
MaxRequests

KeepMaxViews == \A r \in Replica: ViewNumber(r) + 1 <= MaxViews
```

На Листинге 6 приведён основной инвариант абстрактной спецификации — `CommittedLogsPreficesAreEqual`, утверждающий, что у любых двух реплик нет расхождений в закоммиченной части журнала. Если наша модель некорректна и существует исполнение, при котором это нарушается — инструмент TLC покажет нам это.

Далее описаны похожие друг на друга `KeepMaxRequests` и `KeepMaxViews` — они проверяют, что на каждой машине количество клиентских запросов в журнале лога не больше максимума, и что номер `view` у каждой не больше максимального, соответственно.

2.2.5. Fairness и Liveness абстрактной спецификации

Хочется проверять, что при всех допустимых исполнениях наша система совершает в итоге максимум полезной работы, не останавливаясь без действия. В случае алгоритма репликации движков было решено проверить, что модель в

итоге всегда приходит к шагу `Finishing`, описанному в пункте 2.2.3 — то есть, что возможный максимум запросов реплицировался на все машины, и они пришли к одинаковому состоянию.

Для этого существуют `Liveness` свойства модели, описываемые в спецификации и проверяемые вместе с инвариантами. Неплохо поясняются `Liveness` и `Fairness` на уровне концепции в [8]. В языке TLA+ они именно так и реализуются.

На Листинге 7 указано `Liveness` свойство нашей спецификации. Ключевое слово `ENABLED` как раз означает, что действие `Finishing` доступно для выполнения. Значок `<>` означает, что в любом исполнении графа рано или поздно станет верно утверждение записанное, после него. Итого `EventuallyFinished` говорит о том, что любой путь в графе состояний всегда придёт к состоянию, когда все машины полностью реплицированы.

Листинг 7 — `Liveness` свойство абстрактной спецификации

<code>EventuallyFinished == <> (ENABLED Finishing)</code>

Однако есть проблема с проверкой `Liveness` свойств. TLC model checker, строя граф состояний по нашей модели, иногда может делать так называемые `Sluttering` шаги — такие, которые никак не меняют значения наших переменных, словно в мире происходит что-то помимо нашей системы. Они будут мешать нам, поскольку может быть исполнение, где наша система бесконечно долго простаивает. Но такие пути нас не интересуют, мы хотим проверять свойства для системы, которая совершает полезную работу. Для этого существуют `Fairness` ограничения.

На Листинге 8 дано определение `Spec` — традиционное обозначение итоговой спецификации. Состоит из начального состояния и дизъюнкции возможных переходов `Next`. А в `FullSpec` добавлено `Fairness` ограничение, утверждающее, что в графе состояний на любом пути исполнения, если будет бесконечно часто доступно действие `Next` для совершения — то оно будет

совершенно. То есть не будет путей, на которых система будет бесконечно виснуть не совершая полезной работы с точки зрения наших переменных. Это будет гарантировать сам TLC перед проверкой наших свойств.

Также на листинге приведены все утверждения, которые хочется проверить в модели. Ключевое слово THEOREM само по себе ничего не значит для TLA+ или TLC, однако при просмотре спецификации бросается в глаза и намекает, что при проверки графа состояний нужно проверить указанные утверждения.

Первые четыре теоремы описывают инварианты, которые нужно проверить для каждой вершины графа состояний. Они должны выполняться для обычной спеки Spec. А последняя теорема говорит про свойства всех путей в графе, а именно, что если граф соблюдает Fairness ограничение, то реплики в итоге сходятся к конечному состоянию Finishing.

Листинг 8 — Определение абстрактной спецификации с Fairness ограничением и теоремами

```
Spec == Init ∧ [][Next]_vars
FullSpec == Spec ∧ SF_vars(Next)
THEOREM Spec => TypeOK
THEOREM Spec => CommittedLogsPreficesAreEqual
THEOREM Spec => KeepMaxRequests
THEOREM Spec => KeepMaxViews
THEOREM FullSpec => EventuallyFinished
```

2.3. ДЕТАЛЬНАЯ СПЕЦИФИКАЦИЯ

Теперь рассмотрим детальную спецификацию. В ней будут моделироваться сообщения между машинами и логика по их обработке.

2.3.1. Переменные и константы детальной спецификации

В менее высокоуровневой модели сохраняются все те же константы и переменные, что были и в прошлой. Но так же добавляются новые для увеличения детализации системы. На Листинге 9 представлены новые константы и одна переменная.

DownloadChunk, StartDownload, PrepareOk, Commit соответствуют сообщениям из протоколов Download и Normal Operation.

StartViewChange, DoViewChange, StartView — это типы сообщений из View Change протокола.

Все эти константы по смыслу являются такими же модельными значениями, как и статусы реплик, например. Далее в спецификации будет ясно, что хранится в сообщениях этих типов.

Листинг 9 - переменные и константы детальной спецификации

```
\* Message types for processing logs
CONSTANTS DownloadChunk, StartDownload, PrepareOk, Commit

\* Message types for view changing
CONSTANTS StartViewChange, DoViewChange, StartView

\* messages
VARIABLE msgs

vars == <<replicaState, msgs>>
```

2.3.2. Типы в детальной спецификации

На Листинге 10 указан инвариант TypeOK для второй модели. Как видим, переменная replicaState остаётся точно такого же типа, как и в другой спецификации.

Это обязательно, если мы хотим реализовывать абстрактную спецификацию в данной — потому что иначе не будут выполняться шаги той модели при выполнении этой.

В TypeOK записано, что новая переменная `msgs` должна быть подмножеством `Message`. А `Message` — это множество всех допустимых сообщений между репликами. Оно составляется из 7 множеств сообщений разных типов

Листинг 10 — типы детальной спецификации

```

Statuses == {Normal, ViewChange, Recovering}

LogEntry == [type: {RequestBlock}, opNumber: Nat, op: Operation]
           \cup [type: {ViewBlock}, view: Nat]

\* All possible messages
Message == [type: {DownloadChunk}, v: Nat, m: LogEntry, n: Nat, k: Nat, i:
Replica]
           \cup [type: {StartDownload}, v: Nat, n: Nat, src: Replica]
           \cup [type: {PrepareOk}, v: Nat, n: Nat, i: Replica]
           \cup [type: {Commit}, v: Nat, k: Nat]
           \cup [type: {StartViewChange}, v: Nat, i: Replica]
           \cup [type: {DoViewChange}, v: Nat, vv: Nat,
               n: Nat, k: Nat, i: Replica]
           \cup [type: {StartView}, v: Nat, n: Nat, k: Nat]

Send(m) == msgs' = msgs \cup {m}

SendAll(ms) == \w msgs' = msgs \cup ms

TypeOK == \w replicaState \in [
  Replica -> [
    viewNumber: Nat,
    status: Statuses,
    log: Seq(LogEntry),
    downloadReplica: Replica \cup {None},
    commitNumber: Nat
  ]
]
\w msgs \in SUBSET Message

```

- `DownloadChunk` — сообщение с одним элементом лога, указываемым в поле `m`, и позицией этого элемента `n`. Также хранит `v` — номер `view`, в

котором этот элемент был добавлен лидером, k — позицию коммита на момент отправки сообщения и i — отправителя сообщения (чтобы не мешались Download потоки разных реплик). Оно используется и в Normal Operation протоколе вместо Prepare сообщения с новым запросом к движку, полученным мастером, и в Download протоколе, когда новый лидер подкачивает цепочку с другой машины, или когда реплика качает с нового мастера после получения StartView.

- StartDownload — инициация начала скачивания. Хранит v — номер view, n — позицию, с которой начнётся скачивание и src — реплика, с которой будет скачивание (она и примет это сообщение).
- PrepareOk — Отсылается репликой в ответ на DownloadChunk от мастера по Normal Operation протоколу.
- Commit — Посылается периодически лидером, чтобы другие машины узнавали о продвижении позиции коммита и том, что мастер жив.
- StartViewChange — бродкастится (то есть рассылается всем остальным) репликой, перешедшей в новую смену view.
- DoViewChange — шлётся только будущему лидеру от реплики, набравшей информацию о кворуме нод в статусе смены мастера.
- StartView — бродкастит новый лидер, как только полноценно перейдёт в новый view.

2.3.3. Действия в детальной модели

Действия во второй спецификации довольно сильно похожи на действия из первой. Это неудивительно, ведь она должна соответствовать протоколу абстрактной модели, но добавляя детали реализации.

На Листинге 11 приведён пример того же перехода, что и в прошлой спецификации — RecieveClientRequest (получение лидером нового запроса к движку). Как видим, он идентичен своему прототипу в прошлой модели, но добавилась информация о посылке сообщения типа DownloadChunk.

Листинг 11 — пример перехода в детальной модели

```

Send(m) == msgs' = msgs \cup {m}

AddClientRequest(r, m) ==
  ∧ replicaState' = [replicaState EXCEPT ![r].log = Append(@, m)]

RecieveClientRequest(p, op) ==
  ∧ RequestBlockCount(Log(p)) < MaxRequests
  ∧ IsPrimary(p)
  ∧ Status(p) = Normal
  ∧ ~IsDownloading(p)
  ∧ AddClientRequest(p, [type |-> RequestBlock,
                        opNumber |-> OpNumber(p) + 1,
                        op |-> op])
  ∧ Send([type |-> DownloadChunk,
         v |-> ViewNumber(p), m |-> Log(p)'[OpNumber(p) + 1],
         n |-> OpNumber(p) + 1, k |-> CommitNumber(p), i |-> p])

```

Посылка сообщения смоделирована через простое добавления структуры в множество всех сообщений. Мы избавлены от излишних деталей про сетевую инфраструктуру кластера, нам важен только факт отправки и факт получения. Получают реплики сообщения из этого же множества.

Что интересно, таким простым образом уже моделируется различные сбои в системе без дополнительного кода. Например, реплика может сколь угодно долго не получать конкретное сообщение (эмулируя сбой сети или потерю сообщения) — TLC, проходя все пути в графе состояний, рассмотрит и такие исполнения. Или например, если не удалять сообщения из этого множества, то забесплатно эмулируется дублирование сообщения или получения сообщения не той машине — потому что любая реплика может прочитать любое сообщение из множества msgs.

За счёт этого сразу получается и бродкаст сообщений всем нодам, потому что, опять же, любая реплика может читать каждое сообщение, а само сообщение не удаляется после обработки.

На Листинге 12 приведены все разрешённые действия во второй спецификации. Некоторые перечисляются точно так же, как и в первой, например `RecieveClientRequest`. Но некоторым и нужна дополнительная информация, например, для `RecievePrepare` нужно указать помимо реплики ещё и сообщение от лидера.

Листинг 12 — все переходы детальной модели

```

NormalOperationProtocol ==
  √ ∃ r \in Replica, op \in Operation: RecieveClientRequest(r, op)
  √ ∃ r \in Replica, m \in msgs: RecievePrepare(r, m)
  √ ∃ r \in Replica: PrepareOperation(r)
  √ ∃ p \in Replica: AchievePrepareOkFromQuorum(p)
  √ ∃ r \in Replica, m \in msgs: RecieveCommit(r, m)

ViewChangeProtocol ==
  √ ∃ r \in Replica: TimeoutStartViewChanging(r)
  √ ∃ r \in Replica, m \in msgs: RecieveStartViewChange(r, m)
  √ ∃ p \in Replica, m \in msgs: RecieveDoViewChange(p, m)
  √ ∃ r \in Replica: AchieveDoViewChangeFromQuorum(r)
  √ ∃ r \in Replica: SendDownloadChunks(r)
  √ ∃ p \in Replica: MasterDownloadBeforeView(p)
  √ ∃ r \in Replica, m \in msgs: RecieveStartView(r, m)
  √ ∃ r \in Replica: ReplicaDownloadBeforeView(r)

Next == √ NormalOperationProtocol
       √ ViewChangeProtocol
       √ Finishing

```

Остались те же переходы, что были в абстрактном случае, так же добавилось три новых:

- `PrepareOperation` — реплика отсылает `PrepareOk` на элемент лога, если ещё не отвечала на него. Это может происходить, если реплика скачала его по `Download` протоколу после получения `StartView` от нового лидера. Если мы это не будем делать, то можем прийти к дедлоку, когда новый лидер не может закоммитить запрос, так как не собрал достаточно `PrepareOk`, а на репликах запрос уже хранится. Этот переход есть только в

детализированной спецификации, так как он меняет только переменную с сообщениями, а состояние реплик оставляет таким же. В

высокоуровневой модели этого перехода нет, потому что он касается только деталей реализации, а абстрактное описание пропускает этот шаг.

- `RecieveDoViewChange` — получение лидером внезапного сообщения `DoViewChange` с более высоким уровнем `view`, который переводит лидера в новый `view`. Обычное получение лидером сообщений этого вида не моделируется, вместо этого в шаге `AchieveDoViewChangeFromQuorum` сразу рассматривается возможный кворум сообщений, который мог быть получен лидером.
- `SendDownloadChunks` — уведомление любой машины о начале скачивания через `StartDownload` сообщение и отправка ею суффикса журнала лога.

2.3.4. Имплементация абстрактной спецификации

Инварианты, Fairness ограничения и Liveness свойства второй модели будут точно такими же. Но тут стоит вспомнить, что мы хотим проверить выполнимость абстрактной спецификации при выполнении детализированной модели. Если это успешно произойдёт — то модель с сообщениями автоматически унаследует все свойства высокоуровневой и отдельно их проверять не нужно будет.

Для этого в модуле с сообщениями можно создать экземпляр другого модуля, как на Листинге 13.

Листинг 13 — создание экземпляра первого модуля во втором

```
VRNoMsgs == INSTANCE VR_without_message
THEOREM Spec => VRNoMsgs!Spec
```

Через ключевое слово `THEOREM` показываем, что хотим проверять соблюдение спецификации абстрактного модуля во время исполнения нашей

спецификации. В будущем мы укажем это при настройках TLC модели. Что важно, мы указываем *VRNoMsgs!Spec*, а не *VRNoMsgs!FullSpec*. То есть мы проверяем только выполнение Init состояния и Next переходов прошлого модуля, без Fairness ограничений.

Выводы по главе 2

В этой главе мы рассмотрели два TLA+ модуля с двумя разноуровневыми спецификациями — абстрактной и с сообщениями. Их полный код находится в приложениях к данной работе.

ГЛАВА 3. TLC модели

Для написанных TLA+ спецификаций можно создавать TLC модели [7]. Они настраиваются многочисленными параметрами. Например, можно выбирать режим обработки графа — обход в ширину или режим симуляции. При первом TLC перебирает вершины и рёбра обходом в ширину, постепенно увеличивая диаметр графа и изменяя вершины в очереди, пока вершины не кончатся. Во втором режиме TLC генерирует случайные пути в надежде найти ошибку. Этот режим полезен, чтобы перебирать длинные пути в больших графах состояний, пытаясь найти сложные запутанные исполнения и ошибки в них.

В нашем случае стоит выбрать первый обход, чтобы постепенно перебрать весь граф состояний и проверить свойства на всех путях.

3.1. TLC модель абстрактной спецификации

Вначале рассмотрим TLC модель первого модуля

3.1.1. Создание абстрактной модели

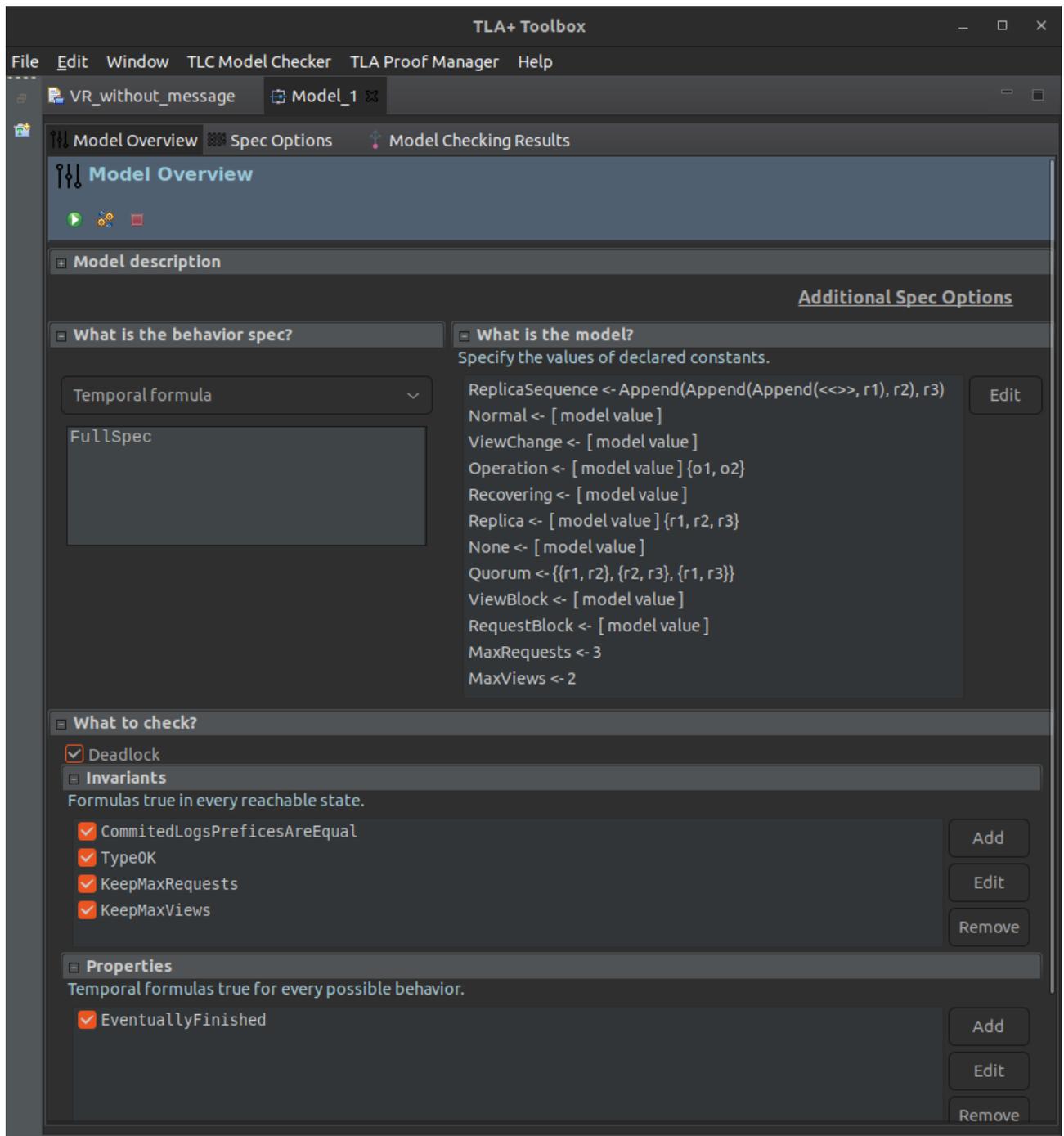


Рисунок 1 — Создание TLC модели абстрактного модуля

На Рисунке 1 изображён интерфейс программы Toolbox, полезной для написания TLA+ спецификаций и запуска TLC моделей. В первом модуле создаём модель. На рисунке видны примеры настроек.

Слева сверху выбирается “behavior spec” — спецификация, по которой генерируется граф. В нашем случае это FullSpec, в которой объединены начальное состояние, разрешённые шаги и Fairness, чтобы корректно работали Liveness свойства.

Правее видны все константы модуля. Для них нужны выбрать значения. Многие из них заполняются просто модельным значением (*[model value]*), которое означает, что у константы будет уникальное для модели значение, не равное ни чему. Это актуально в нашем случае для статусов реплики, None и типов блоков в журналах машин. Operation и Replica заполняются множествами модельных значений $\{o1, o2\}$ и $\{r1, r2, r3\}$. Для этих значений верно то же самое. Quorum и ReplicaSequence заполняются множеством и списком из модельных значений, соответствующих репликам. А MaxRequests и MaxViews равны просто числам.

Именно эти числа отвечают за ограничения пространства состояний. На рисунке приведены соответствующие значения 3 и 2. В этом случае будут перебираться состояния с не более, чем 3 запросами к движкам в логе у каждой реплики, и view не большими, чем 1. То есть будут реплицироваться 3 клиентских команды и будет максимум 1 смена лидера. Однако в процессе смены view некоторые команды могут потеряться и не перейти в финальную цепочку (например, если первый лидер не будет участвовать в новых выборах). В таком случае будут генерироваться новые команды на новом лидере и под конец их всё равно наберётся 3.

Далее указывается галочка около слова Deadlock. Это означает, что TLC сообщит нам, если найдёт путь, приводящий систему в состояние, где она не может сделать ни одно допустимое в Next действие. Нам это важно проверять, потому что мы хотим, чтобы все реплики в итоге приходили в Finishing состояние, где все журналы полностью закоммичены и состояния машин идентичны. Это состояние не будет считаться дедлоком, потому что в нём

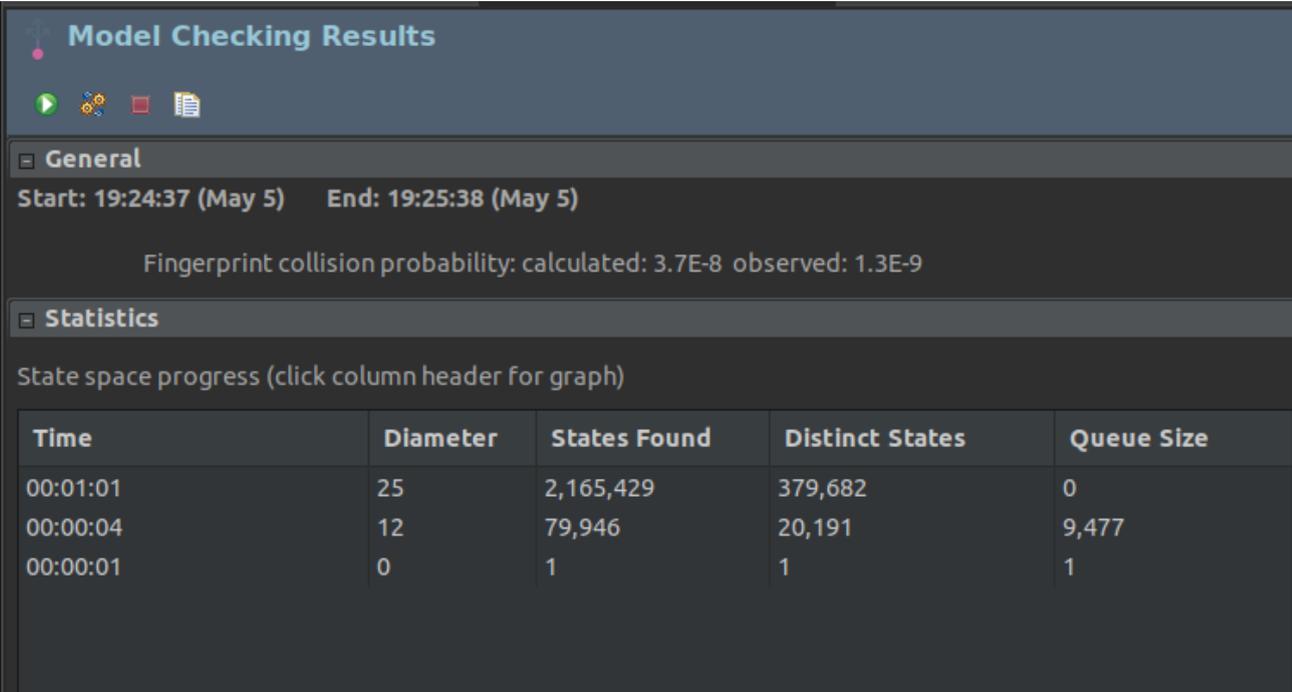
всегда можно выполнить шаг `Finishing`, не меняющий переменную `replicaState`. Для этого мы его и добавляли.

Ниже перечисляются инварианты модели. Каждый инвариант — это условие, которое будет проверяться для каждой вершины графа состояний. Это и есть теоремы, что обсуждались в конце пункта 2.2.5

После этого указываются свойства, которые должна иметь система. Они уже проверяются не на отдельных вершинах, а на целых путях. В нашем случае это `EventuallyFinished`, утверждающий, что мир рано или поздно должен перейти в состояние `Finishing`.

3.1.2. Запуск абстрактной модели

После настройки модель её можно запустить и далее ждать неопределённое время. TLC периодически пишет текущую статистику — размеры графа и количество переходов по действиям.



The screenshot shows a window titled "Model Checking Results" with a dark theme. It contains two main sections: "General" and "Statistics".

General
 Start: 19:24:37 (May 5) End: 19:25:38 (May 5)
 Fingerprint collision probability: calculated: 3.7E-8 observed: 1.3E-9

Statistics
 State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:01:01	25	2,165,429	379,682	0
00:00:04	12	79,946	20,191	9,477
00:00:01	0	1	1	1

Рисунок 2 — Результат запуска абстрактной модели

На рисунке 2 приведён результат запуска модели, созданной в предыдущем пункте. Длился 1 минуту. Не видим сообщений об ошибках,

следовательно запуск успешен. В таблице указывается изменяемая со временем статистика об обработке графа — текущий диаметр, количество состояний (вернее, рёбер), количество различных состояний (вершин графа), размер bfs очереди. Запуск шёл минуту. Более детальные модели будут работать дольше. Помимо этого, при больших ограничениях пространства состояний запуск тоже будет идти дольше.

Sub-actions of next-state (at 00:01:01)				
Module	Action	Location	States Found	Distinct States
VR_without_message	RecieveStartViewChange	line 190, col 1 to line 190, col 25	282,436	0
VR_without_message	Finishing	line 291, col 1 to line 291, col 9	40	0
VR_without_message	Init	line 57, col 1 to line 57, col 4	1	1
VR_without_message	MasterDownloadBeforeVi	line 234, col 1 to line 234, col 27	10,722	3
VR_without_message	RecieveClientRequest	line 122, col 1 to line 122, col 27	207,382	20,251
VR_without_message	AchieveDoViewChangeFro	line 202, col 1 to line 202, col 32	40,484	20,752
VR_without_message	ReplicaDownloadBeforeVi	line 271, col 1 to line 271, col 28	144,532	33,374
VR_without_message	RecievePrepare	line 131, col 1 to line 131, col 17	236,064	43,778
VR_without_message	TimeoutStartViewChangin	line 182, col 1 to line 182, col 27	170,334	47,261
VR_without_message	RecieveStartView	line 257, col 1 to line 257, col 19	246,180	66,994
VR_without_message	RecieveCommit	line 163, col 1 to line 163, col 16	593,243	72,339
VR_without_message	AchievePrepareOkFromQt	line 153, col 1 to line 153, col 29	234,011	74,929

Рисунок 3 — статистика переходов абстрактной модели

На Рисунке 3 приведено статистика переходов модели, которая тоже показывается после запуска. Тут для каждого действия указано количество состояний, найденным при переходе по нему и количество уникальных таких. По сути тоже количество рёбер с таким переходом и количество уникальных вершин. Например, для RecieveStartViewChange видим 282436 состояний, но 0 уникальных. Это объясняется тем, что в абстрактной спецификации все переходы по этому действию доступны также и по действию TimeoutStartViewChanging, у которого уже есть 47261 уникальных. И по нему они перебираются раньше в обходе в ширину.

С Finishing интересная ситуация, у него мало состояний, потому что действительно очень редко реплики приходят к итоговому консенсусу по сравнению с количеством шагов до этого. И нет уникальных, потому что он вообще не меняет переменных. Это просто петля в ориентированном графе состояний.

Если бы по каким-то другим переходам не было найдено состояний, это бы вызвало вопросы о корректности нашей модели.

Но самое важное, что мы получили из этого запуска — это отсутствие ошибок, значит, все наши инварианты и свойства работают, реплики консистентны по закоммиченным частям логов и в итоге всегда полностью реплицируются. Очень важный результат.

Но важно помнить, что это всё справедливо для очень ограниченного пространства состояний: 3 запроса к движку и 1 смена view. В пункте 3.3 с результатами приведена таблица проверенных параметров.

3.2. TLC МОДЕЛЬ ДЕТАЛИЗИРОВАННОЙ СПЕЦИФИКАЦИИ

Теперь нужно проделать то же самое для второй спецификации, но проверить нужно будет другое свойство.

3.2.1. Создание детализированной модели

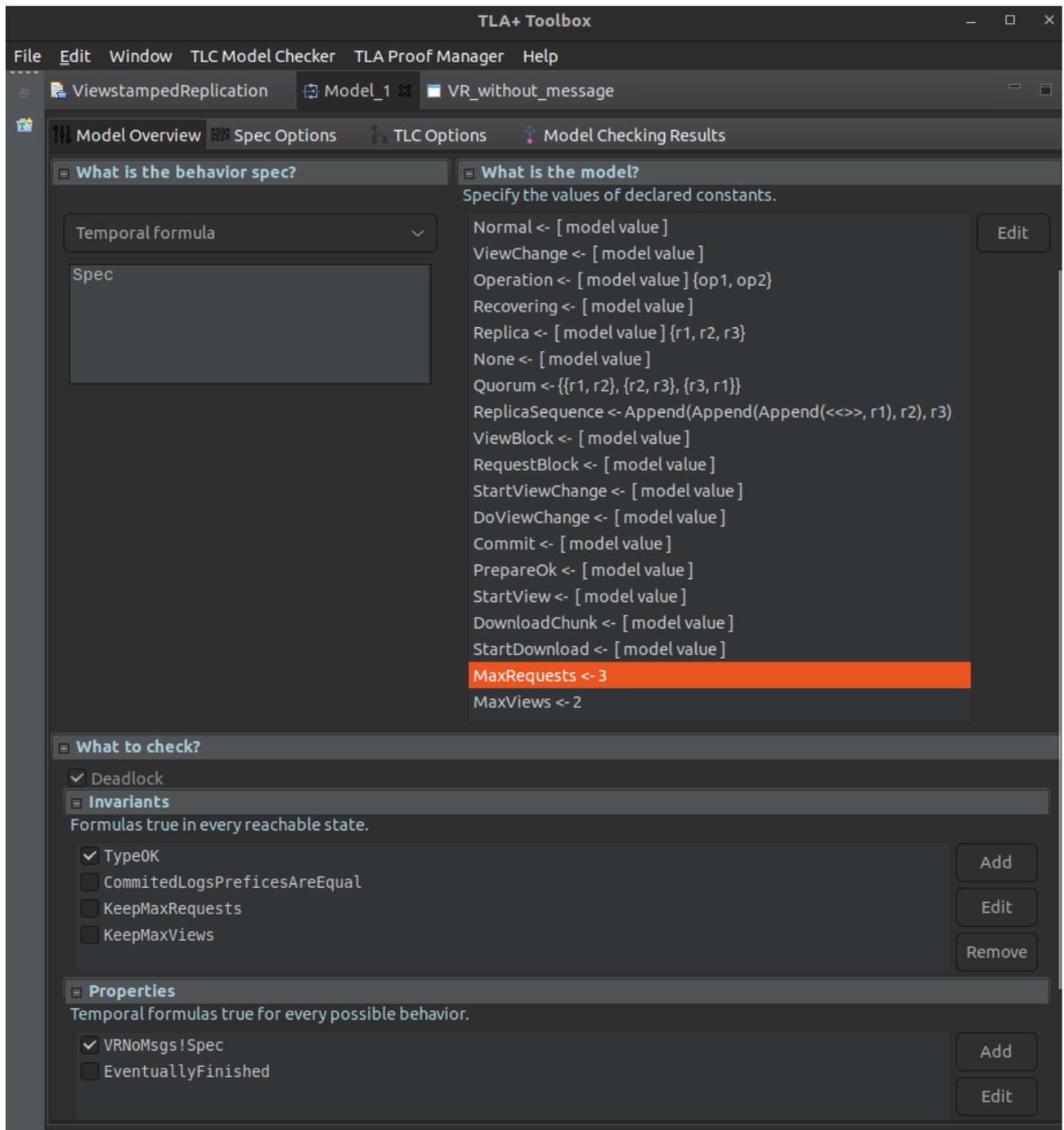


Рисунок 4 — создание детализированной модели

Создание модели для второго модуля выглядит так же, однако констант больше. Добавляются типы сообщений. Для них всех достаточно выбрать модельные значения.

Дедлок тоже нужно проверять, иначе система может зависать, а мы это не отловим.

Все инварианты проверять необязательно, так как они уже верны для абстрактной спеки. Достаточно отметить TypeOK, так как в нём появилось утверждение про переменную msgs.

EventuallyFinished тоже отмечать не нужно, поскольку оно верно для абстрактной модели. Обязательно нужно выбрать VRNoMsgs!Spec, поскольку в этом цель запуска — проверить имплементацию другой спецификации.

3.2.2. Запуск детализированной модели

Model Checking Results

Start: 20:10:28 (May 5) Last checkpoint: 20:40:33 (May 5) End: 21:05:42 (May 5)

Fingerprint collision probability: calculated: 7.7E-5 observed: 2.0E-5

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:55:13	27	98,726,226	17,577,424	0
00:55:06	27	98,617,304	17,573,020	30,018
00:54:06	27	97,145,107	17,429,522	261,313
00:53:06	26	95,614,508	17,266,538	474,809
00:52:06	26	94,017,412	17,073,535	648,748
00:51:06	25	92,464,902	16,866,594	781,955
00:50:06	25	90,754,761	16,619,131	879,291
00:49:06	25	89,122,869	16,425,159	1,067,139
00:48:06	24	87,421,945	16,188,154	1,187,788
00:47:05	24	85,765,917	15,942,881	1,272,770
00:46:05	24	83,975,055	15,659,777	1,322,847
00:45:05	24	82,250,816	15,423,433	1,458,138
00:44:05	23	80,537,300	15,187,347	1,592,305
00:43:05	23	78,852,602	14,921,030	1,634,677
00:42:05	23	77,048,277	14,635,205	1,685,718
00:41:05	23	75,259,118	14,338,378	1,707,339
00:40:05	23	73,471,752	14,074,096	1,800,851
00:39:05	23	71,799,198	13,821,083	1,879,243
00:38:05	22	69,983,597	13,543,912	1,959,759
00:37:05	22	68,095,627	13,223,590	1,967,445
00:36:05	22	66,159,561	12,900,687	1,987,281
00:35:05	22	64,305,903	12,584,269	1,997,119
00:34:05	22	62,329,842	12,274,781	2,053,579
00:33:05	22	60,349,997	11,957,437	2,101,576
00:32:05	21	58,414,100	11,646,706	2,156,321
00:31:05	21	56,603,104	11,326,711	2,143,042

Рисунок 5 — запуск модели с сообщениями

На Рисунке 5 виден результат запуска модели. Это заняло 55 минут. Проверка предыдущей модели заняла всего 1 минуту. Хотя ограничения на пространство состояний одинаковые.

Становится наглядно видно, как увеличение детализации спецификации значительно влияет на размер модели. Диаметр графа увеличился не сильно, всего на 2 — стал 27 вместо 25. Зато рёбер 98726226 вместо 2165429 — примерно в 46 раз больше. А вершин стало 17577424 вместо 379682, что тоже увеличение примерно в 46 раз.

Sub-actions of next-state (at 00:55:13)				
Module	Action	Location	States Found	Distinct States
ViewstampedReplication	Finishing	line 348, col 1 to line 348, col 9	328	0
ViewstampedReplication	Init	line 82, col 1 to line 82, col 4	1	1
ViewstampedReplication	PrepareOperation	line 159, col 1 to line 159, col 19	4,044	153
ViewstampedReplication	NormalOperationProtoco	line 367, col 8 to line 367, col 57	872,832	2,534
ViewstampedReplication	AchievePrepareOkFromQi	line 169, col 1 to line 169, col 29	392,169	3,391
ViewstampedReplication	NormalOperationProtoco	line 370, col 8 to line 370, col 56	2,084,412	51,898
ViewstampedReplication	ViewChangeProtocol	line 375, col 8 to line 375, col 62	339,900	113,300
ViewstampedReplication	ReplicaDownloadBeforeVi	line 316, col 1 to line 316, col 28	138,624	127,085
ViewstampedReplication	MasterDownloadBeforeVi	line 271, col 1 to line 271, col 27	1,100,976	1,006,770
ViewstampedReplication	TimeoutStartViewChangin	line 195, col 1 to line 195, col 27	3,078,270	1,166,655
ViewstampedReplication	RecieveClientRequest	line 134, col 1 to line 134, col 27	4,047,796	1,180,672
ViewstampedReplication	AchieveDoViewChangeFro	line 234, col 1 to line 234, col 32	2,379,300	1,570,508
ViewstampedReplication	ViewChangeProtocol	line 374, col 8 to line 374, col 65	54,959,930	3,195,806
ViewstampedReplication	ViewChangeProtocol	line 379, col 8 to line 379, col 59	6,351,952	3,447,611
ViewstampedReplication	SendDownloadChunks	line 259, col 1 to line 259, col 21	22,975,692	5,711,040

Рисунок 6 — статистика переходов детализированной модели

На Рисунке 6 приведено количество рёбер по переходам. По статистике видно, что очень много действий отводится под `SendDownloadChunks`, например. Можно подумать в сторону каких-то оптимизаций, чтобы сократить граф.

Но самое главное это то, что ошибок не было — значит модель с сообщениями, сделанная по алгоритму в Базе Данных ВК, действительно реализует абстрактную на данном пространстве состояний. Поэтому для неё верны её свойства — консистентность логов и итоговая сходимость реплик в одно состояние с тремя реплицированными и закоммиченными запросами к движкам.

3.3. РЕЗУЛЬТАТЫ

В Таблицах 1 и 2 представлены результаты запусков двух моделей с разными параметрами MaxRequests и MaxViews. Во всех запусках количество реплик было 3.

Таблица 1 — результаты для абстрактной модели

Кол-во запросов к движку	Кол-во view	Время запуска	Диаметр графа	Рёбра	Вершины
6	1	00:01:15	28	3 643 293	352 523
4	2	00:08:41	29	15 522 413	2 535 970
2	3	00:01:57	28	3 719 933	662 780
1	4	00:01:11	30	1 613 677	288 331
0	5	00:00:09	31	177 139	32 532

Таблица 2 — результаты для детализированной модели

Кол-во запросов к движку	Кол-во view	Время запуска	Диаметр графа	Рёбра	Вершины
5	1	00:19:00	26	39 725 240	6 905 386
3	2	00:55:13	27	98 726 226	17 577 424
1	3	00:05:31	30	9 936 790	1 876 202
0	4	00:00:09	28	207 661	52 976
3	3	14:01:15	более 26	более 1 291 484 763	более 308 926 479

В последней строке Таблицы 2 указана статистика неоконченного запуска модели с сообщениями. Запуск был приостановлен в связи с нехваткой ресурсов

и длительностью более 14 часов. Он не позволяет проверить полностью всё пространство состояний с 3 запросами и 3 view. Но он добавляет уверенности в корректности модели на довольно большом графе.

На Рисунке 7 показана поминутная статистика в течение последнего получаса. Видно, что очередь вершин в обходе в ширину только увеличивалась, что говорит о большом размере непросмотренного графа. Если бы очередь уменьшалась — это бы дало понять, что проверка скоро завершится.

Model Checking Results (model checking is in progress)

General
Start: 23:51:43 (May 5) Last checkpoint: 13:30:47 (May 6)

Statistics
State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
14:01:15	26	1,291,484,763	308,926,479	94,036,371
14:00:15	26	1,289,874,149	308,582,829	93,961,752
13:59:15	26	1,288,250,558	308,262,061	93,914,409
13:57:59	26	1,286,384,912	307,858,683	93,806,504
13:56:59	26	1,285,088,104	307,577,880	93,732,591
13:55:59	26	1,283,767,273	307,293,725	93,663,028
13:54:59	26	1,282,411,764	306,999,932	93,600,276
13:51:51	26	1,278,218,002	306,054,128	93,348,796
13:50:51	26	1,276,927,367	305,776,096	93,281,290
13:49:51	26	1,275,672,670	305,475,829	93,181,374
13:48:51	26	1,274,307,436	305,143,838	93,097,306
13:45:55	26	1,270,142,040	304,143,051	92,845,831
13:44:55	26	1,268,820,551	303,826,448	92,742,555
13:43:55	26	1,267,353,846	303,442,368	92,614,980
13:42:55	26	1,265,927,973	303,082,514	92,495,240
13:40:04	26	1,261,923,276	302,087,783	92,171,656
13:39:04	26	1,260,532,766	301,738,401	92,059,424
13:38:03	26	1,259,094,906	301,378,778	91,943,692
13:37:03	26	1,257,677,192	301,059,244	91,876,623
13:35:06	26	1,255,067,922	300,438,977	91,706,235
13:34:06	26	1,253,645,262	300,076,958	91,585,933
13:33:06	26	1,252,287,007	299,740,789	91,476,823
13:32:06	26	1,251,045,209	299,430,582	91,369,145
13:30:10	26	1,248,273,888	298,734,450	91,143,453
13:29:10	26	1,246,868,055	298,373,148	91,024,816
13:28:10	26	1,245,467,083	298,042,303	90,934,961
13:27:10	26	1,243,906,115	297,770,337	90,938,576
13:25:49	26	1,241,859,478	297,374,306	90,888,086

Рисунок 7 — статистика неоконченного 14 часового запуска модели с сообщениями.

Выводы по главе 3

Были созданы TLC модели по спецификациям из прошлой главы. Модель без сообщений успешно сохраняет свойство EventuallyFinishing и консистентность логов на некоторых проверенных пространствах состояний.

Модель с сообщениями успешно реализует первую модель на некоторых пространствах состояний, следовательно тоже сохраняет эти свойства.

ЗАКЛЮЧЕНИЕ

В этой работе был рассмотрен алгоритм репликации движков в базе данных ВК. Также, был описан исходный алгоритм Viewstamped Replication и оптимизации разработчиков.

В Главе 1 было рассказано про язык TLA+ и верификацию систем через него. Обозначены общие проблемы TLA+ спецификаций и TLC моделей. Поставлена задача верификации алгоритма репликации движков через двухуровневую модель.

В Главе 2 разработана абстрактная модель, задающая контракт на взаимодействие реплик. Сформулированы свойства и инварианты для проверки их на графе состояний. Также, разработана модель с сообщениями, успешно реализующая первую на некоторых ограниченных пространствах состояний. По транзитивности детализированная спецификация автоматически сохраняет свойства абстрактной.

В Главе 3 произведены запуски на некоторых ограниченных пространствах состояний, которые подтверждают корректность спецификации.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. КАК УСТРОЕНЫ БАЗЫ ДАННЫХ ВКОНТАКТЕ [ЭЛЕКТРОННЫЙ РЕСУРС]. — 11 ДЕКАБРЯ 2018. — URL: https://vk.com/@brown_room-database-vk.
2. Архитектура серверов и баз данных ВКонтакте [ЭЛЕКТРОННЫЙ РЕСУРС]. — 12 ДЕКАБРЯ 2018. — URL: https://vk.com/@brown_room-servers-and-database-in-vk.
3. Leslie Lamport. ACM Transactions on Computer Systems 16. — 2 May 1998. — P. 133-169.
4. Diego Ongaro, John Ousterhout. In Search of an Understandable Consensus Algorithm (Extended Version). — 19 June 2014. — URL: <https://raft.github.io/raft.pdf>.
5. Barbara Liskov, James Cowling. Viewstamped Replication Revisited — July 2012. — URL: <https://pmg.csail.mit.edu/papers/vr-revisited.pdf>.
6. Leslie Lamport. Specifying Concurrent Systems with TLA+. — 3 March 1999 — URL: <https://lamport.azurewebsites.net/pubs/lamport-spec-tla-plus.pdf>.
7. Leslie Lamport. Specifying Systems. — 18 June 2002. — <https://lamport.azurewebsites.net/tla/book-21-07-04.pdf>. — P. 221-264.
8. Косяков Михаил Сергеевич. Введение в распределённые вычисления. — URL: <https://books.ifmo.ru/file/pdf/1551.pdf>. — P. 51-52.