Telegram Open Network Blockchain

Николай Дуров

8 февраля 2020

Аннотация

Целью этого текста является предоставление подробного описания блокчейна Telegram Open Network (TON).

Знакомство

Этот документ содержит подробное описание блокчейна TON, включая его точный формат блока, условия действия, сведения о вызове виртуальной машины TON (TVM), процесс создания смарт-контракта и криптографические подписи. В этом отношении он является продолжением технического документа TON (см. [3]), поэтому мы свободно используем терминологию, представленную в этом документе.

В главе 1 представлен общий обзор блокчейна TON и принципов его проектирования, с особым вниманием к введению условий совместимости и валидности и реализации гарантий доставки сообщений. Более подробная информация, такая как схемы TL-B, которые описывают сериализацию всех необходимых структур данных в деревья или коллекции («мешки») ячеек, представлена в последующих главах, кульминацией которых является полное описание макета блоков TON Blockchain (shardchain и masterchain) в главе 5.

Подробное описание криптографии эллиптической кривой, используемой для подписи блоков и сообщений, также доступной через примитивы TVM, приведено в Приложении A. Сам TVM описан в отдельном документе (ср. [4]).

Некоторые темы намеренно были исключены из этого документа. Одним из них является византийский отказоустойчивый протокол (BFT), используемый валидаторами для определения следующего блока мастерчейна или шардчейна; эта тема оставлена для предстоящего документа, посвященного сети TON. И хотя этот документ описывает точный

формат блоков TON Blockchain и обсуждает условия валидности блокчейна и сериализованные доказательства недействительности, [1] он не предоставляет никаких подробностей о сетевых протоколах, используемых для распространения этих блоков, кандидатов на блоки, сопоставленных блоков и доказательств недействительности.

Аналогичным образом, этот документ не предоставляет полный исходный код смарт-контрактов masterchain, используемых для выбора валидаторов, изменения настраиваемых параметров или получения их текущих значений, или наказания валидаторов за их неправильное поведение, хотя эти смарт-контракты составляют важную часть общего состояния блокчейна и нулевого блока мастерчейна. Вместо этого в этом документе описывается местоположение этих смарт-контрактов и их формальных интерфейсов. [2] Исходный код этих смарт-контрактов будет предоставлен отдельно в виде загружаемых файлов с комментариями.

Обратите внимание, что текущая версия этого документа описывает предварительную тестовую версию блокчейна TON; некоторые незначительные детали могут измениться перед запуском на этапах разработки, тестирования и развертывания.

[1] По состоянию на август 2018 года этот документ не содержит подробного описания сериализованных доказательств недействительности, поскольку они, вероятно, значительно изменятся в ходе разработки программного обеспечения валидатора. Обсуждаются только общие принципы проектирования условий согласованности и серийных доказательств недействительности.

[2] Он не включен в настоящий вариант настоящего документа, но будет представлен в отдельном добавлении к будущему пересмотренному варианту.

Содержание

1 Обзор

В этой главе представлен обзор основных функций и принципов проектирования блокчейна TON. Более подробная информация по каждой теме приводится в последующих главах.

1.1 Все – мешок ячеек

Все данные в блоках и состоянии TON Blockchain представлены в виде совокупности *ячеек* (ср. [3, 2.5]). Поэтому эта глава начинается с общего обсуждения клеток.

1.1.1. TBM-ячейки. Напомним, что блокчейн TON, как и виртуальная машина TON (TVM; см. [4]), представляет все постоянно хранящиеся данные в виде *коллекции* или *мешка* так называемых *ячеек. Каждая ячейка состоит из 1023 бит данных и до четырех ссылок на другие ячейки. Циклические клеточные ссылки не допускаются, поэтому клетки обычно организованы в деревья клеток, а точнее направленные ациклические графы (DAG) клеток.* [3] Любое значение абстрактного алгебраического (зависимого) типа данных может быть представлено (сериализовано) в виде дерева ячеек. Точный способ представления значений абстрактного типа данных в виде дерева ячеек выражается с помощью *схемы TL-B.*[4] Более подробное обсуждение различных видов клеток можно найти в [4, 3.1].

1.1.2. Применение к TON Blockchain блоков и состояния. Вышесказанное особенно применимо к блокам и состоянию блокчейна TON, которые также являются значениями определенных (довольно запутанных) зависимых алгебраических типов данных. Поэтому они сериализуются по различным схемам TL-B (которые постепенно представлены в этом документе) и представлены в виде коллекции или мешка клеток.

[3] Полностью идентичные ячейки часто идентифицируются в памяти и в дисковом хранилище; именно по этой причине деревья клеток прозрачно трансформируются в DAG клеток. С этой точки зрения группа обеспечения доступности баз данных — это просто оптимизация хранилища базового дерева ячеек, не имеющая значения для большинства соображений.

[4] [4, 3.3.3–4], где приведен и объяснен пример в ожидании более полной ссылки

1.1.3. Компоновка одной ячейки. Каждая отдельная ячейка состоит из 1023 бит данных и до четырех ссылок на другие ячейки. Когда ячейка хранится в памяти, ее точное представление зависит от реализации. Однако существует стандартное представление ячеек, полезное, например, для сериализации ячеек для хранения файлов или передачи по сети. Это "стандартное представление" или "типовая форма стандартов" CellRepr(c) ячейки *c* состоит из следующего:

- На первом месте *стоят два дескрипторных* байта, иногда обозначаемые *d1* и *d2. Первый из этих байтов d1* равен (в простейшем случае) количеству ссылок $0 \le r \le 4$ в ячейке. Второй дескриптор байт *d2* кодирует битовую длину l части данных ячейки следующим образом: первые семь битов *d2* равны $\lceil l/8 \rceil$, количеству полных байтов данных, присутствующих в ячейке, в то время как последний бит *d2* является *тегом завершения, равным единице, если l* не делится на восемь. Следовательно

$$d2 = 2\lceil l/8 \rceil + [l \bmod 8 \neq 0] = \lceil l/8 \rceil + \lceil l/8 \rceil \qquad (1)$$

  где $[A]$ равно единице, когда условие A истинно, и нулю в противном случае.

- Далее следуют байты данных $\lceil l/8 \rceil$. Это означает, что l биты данных ячейки разделены на группы по восемь, и каждая группа интерпретируется как 8-битное целое число с большим порядком байтов и хранится в байте. Если l не делится на восемь, к битам данных добавляется один двоичный единица и подходящее количество двоичных нулей (до шести), и устанавливается тег завершения (наименее значимый бит дескриптора *байта d2).*

- Наконец, далее следуют ссылки r на другие ячейки. Каждая ссылка обычно представлена 32 байтами, содержащими хэш sha256 ячейки, на которую указывает ссылка, вычисляемый, как описано ниже в 1.1.4.

Таким образом, стандартное представление CellRepr(c) ячейки *c* с *l* битами данных и *ссылками r* составляет $2 + \lceil l/8 \rceil + \lceil l/8 \rceil + 32r$ байт длиной.

1.1.4. Хэш sha256 ячейки. Хэш sha256 ячейки *c* рекурсивно определяется как sha256 стандартного представления CellRepr(c) рассматриваемой ячейки:

$$\text{Hash (c)} := \text{sha256(c)} := \text{sha256 (CellRepr}^{(c)}) \quad (2)$$

Поскольку циклические ссылки на ячейки не допускаются (отношения между всеми ячейками должны составлять направленный ациклический граф или DAG), хэш sha256 ячейки всегда четко определен.

Кроме того, поскольку sha256 молчаливо считается устойчивым к столкновениям, мы предполагаем, что все клетки, с которыми мы сталкиваемся, полностью определяются их хешами. В частности, клеточные ссылки ячейки *c* полностью определяются хэшами ссылочных ячеек, содержащимися в стандартном представлении CellRepr(c).

1.1.5. Экзотические клетки. Помимо *обычных* клеток (также называемых *простыми* или *данными*), рассмотренных до сих пор, ячейки других типов, называемые *экзотическими клетками, иногда появляются в реальных представлениях блоков TON Blockchain и других структур данных. Их представление несколько отличается; они отличаются наличием первого дескрипторного байта d1* ≥ 5 (ср. [4, 3.1]).

1.1.6. Внешние опорные ячейки. *(Внешние) ссылочные ячейки, которые содержат 32-байтовый sha256(c)* «истинной» ячейки данных *c* вместо самой ячейки данных, являются одним из примеров экзотических ячеек. Эти ячейки могут быть использованы при сериализации пакета ячеек, соответствующего блоку TON Blockchain, чтобы ссылаться на ячейки данных, отсутствующие в сериализации самого блока, но предположительно присутствующие где-то еще (например, в предыдущем состоянии блокчейна).

1.1.7. Прозрачность опорных ячеек в отношении большинства операций. Большинство клеточных операций не наблюдают никаких эталонных клеток или других «экзотических» видов клеток; они видят только ячейки данных, причем любая ссылочная ячейка прозрачно заменяется ячейкой, на которую ссылается. Например, когда *прозрачный* хэш ячейки Hash^{b}(c) вычисляется рекурсивно, хэш эталонной ячейки устанавливается равным

хэшу упомянутой ячейки, а не хэшу стандартного представления эталонной ячейки.

1.1.8. Прозрачный хэш и хэш представления ячейки. Таким образом, $\text{sha256}^b(c) = \text{Hash}^b(c)$ является *прозрачным хэшем* ячейки *c* (или дерева ячеек, укорененных в *c).*

Однако иногда нам нужно рассуждать о точном представлении дерева клеток, присутствующих в блоке. С этой целью определяется *хэш представления* $\text{Hash}^b(c)$, который не является прозрачным по отношению к эталонным ячейкам и другим экзотическим типам ячеек. Мы часто говорим, что хэш представления *c* является «хэшем» *c, потому что это наиболее часто используемый хэш ячейки.*

1.1.9. Использование репрезентативных хэшей для подписей. Подписи являются отличным примером применения хэшей представления. Например:

- Валидаторы подписывают хэш представления блока, а не только его прозрачный хэш, потому что им необходимо подтвердить, что блок содержит необходимые данные, а не только некоторые внешние ссылки на них.

- Когда внешние сообщения подписываются и отправляются сторонами вне цепочки (например, клиентами-людьми, использующими приложение для инициирования транзакций блокчейна), если внешние ссылки могут присутствовать в некоторых из этих сообщений, именно хэши представления сообщений должны быть подписаны.

1.1.10. Высшие хэши ячейки. Помимо прозрачных и репрезентативных хэшей ячейки *c, существует последовательность высших хэшей* $\text{Hash}_i(c)$, $i = 1,2,\ldots$ может быть определено, что в конечном итоге стабилизируется при $\text{Hash}_\infty(c)$. (Более подробную информацию можно найти в [4, 3.1].)

1.2 Основные компоненты блока и состояние блокчейна

В этом разделе кратко описываются основные компоненты блока и состояния блокчейна, не вдаваясь в детали.

1.2.1. Парадигма бесконечного шардинга (ISP), применяемая к блоку и состоянию блокчейна. Напомним, что в соответствии с парадигмой Бесконечного Шардинга каждая учетная запись может рассматриваться как лежащая в отдельной «цепочке учетных записей», а (виртуальные) блоки этих связей счетов затем группируются в блоки шардчейнов для целей эффективности. В частности, состояние шардчейна состоит, грубо говоря, из состояний всех его «счетных цепей» (т. е. всех счетов, назначенных ему); точно так же блок шардчейна по существу состоит из коллекции виртуальных «блоков» для некоторых учетных записей, назначенных шардчейну. [5]Мы можем резюмировать это следующим образом:

$$ShardState \approx Hashmap(n, AccountState) \quad (3)$$

$$ShardBlock \approx Hashmap(n, AccountBlock) \quad (4)$$

где n — битовая длина *account_id, a Hashmap(n,X)* описывает частичную карту $\mathbf{2^n} \rightarrow X$ из битовых строк длиной n в значения типа X.

Напомним, что каждый шардчейн — или, точнее, каждый блок шардчейна [6] — соответствует всем цепочкам учетных записей, которые принадлежат к одной и той же «рабочей цепи» (т. е. имеют одинаковый *workchain_id = w) и имеют account_id*, начинающуюся с одного и того же двоичного префикса s, *так что* (w,s) полностью определяет осколок. Поэтому приведенные выше хэш-карты должны содержать только ключи, начинающиеся с префикса s.

Через мгновение мы увидим, что приведенное выше описание является лишь приближением: состояние и блок шардчейна должны содержать

[5] Если транзакций, связанных со счетом, нет, соответствующий виртуальный блок пуст и опущен в блоке shardchain

[6] Напомним, что TON Blockchain поддерживает *динамическое* шардинг, поэтому конфигурация сегмента может меняться от блока к блоку из-за событий слияния и разделения сегментов. Поэтому мы не можем просто сказать, что каждый шардчейн соответствует фиксированному набору цепей счетов.

некоторые дополнительные данные, которые не разделены в соответствии с *account_id* как предлагается (3).

1.2.2. Разделенная и неразделенная часть шардчейнового блока и состояния. Блок шардчейна и его состояние могут быть разделены на две отдельные части. Части с продиктованной ISP формой (3) будут называться разделенными частями блока и его состоянием, а остальные будут называться *неразделенными* частями.

1.2.3. Взаимодействие с другими блоками и внешним миром. Условия глобальной и локальной согласованности. Неразделенные части блока шардчейна и его состояние в основном связаны с взаимодействием этого блока с некоторыми другими «соседними» блоками. Условия глобальной согласованности блокчейна в целом сводятся к условиям внутренней согласованности отдельных блоков сами по себе, а также к внешним локальным условиям согласованности между определенными блоками (см. 1.3).

Большинство из этих локальных условий согласованности связаны с пересылкой сообщений между различными цепочками сегментов, транзакциями, включающими более одного шардчейна, и гарантиями доставки сообщений. Однако другая группа условий локальной согласованности связывает блок с его непосредственными предшественниками и преемниками внутри шардчейна; например, начальное состояние блока обычно должно совпадать с конечным состоянием его непосредственного предшественника.[7]

1.2.4. Входящие и исходящие сообщения блока. Наиболее важными компонентами нерасщепленной части блока шардчейна являются следующие:

- *InMsgDescr* — описание всех сообщений, «импортированных» в этот блок (т.е. либо обработанных транзакцией, включенной в блок, либо

[7] Это условие применяется, если существует точно один непосредственный предшественник (т.е. если событие слияния шардчейна не произошло непосредственно перед рассматриваемым блоком); в противном случае это состояние становится более запутанным.

переадресованных в очередь вывода, в случае транзитного сообщения, движущегося по пути, продиктованному Hypercube Routing).

- *OutMsgDescr* — описание всех сообщений, «экспортированных» или «сгенерированных» блоком (т.е. либо сообщений, сгенерированных транзакцией, включенной в блок, либо транзитных сообщений с назначением, не принадлежащим текущему shardchain, пересылаемого из *InMsgDescr).*

1.2.5. Заголовок блока. Другим неразделенным компонентом блока shardchain является *заголовок блока, который содержит общую информацию, такую как* (w,s) (т.е. *workchain_id* и общий двоичный префикс всех *account_ids, назначенных текущему shardchain), порядковый номер блока* (определяемый как наименьшее неотрицательное целое число, большее, чем порядковые номера его предшественников), *логическое время и генерация unixtime. Он также содержит хэш непосредственного предшественника блока (или двух его непосредственных предшественников в случае предшествующего события слияния shardchain), хэши его начального и конечного состояний (т. е. состояний shardchain непосредственно до и сразу после обработки текущего блока) и хэш самого последнего блока masterchain, известного на момент генерации блока shardchain.*

1.2.6. Подписи Валидатора, подписанные и неподписанные блоки. Блок, описанный до сих пор, является *беззнаковым блоком; он генерируется полностью и рассматривается валидаторами как единое целое. Когда валидаторы в конечном итоге подписывают его,* создается подписанный блок, состоящий из беззнакового блока вместе со списком подписей валидатора (определенного хэша представления беззнакового блока, см. 1.1.9). Этот список подписей также является неразделенным компонентом (подписанного) блока; однако, поскольку он находится за пределами беззнакового блока, он несколько отличается от других данных, хранящихся в блоке.

1.2.7. Очередь исходящих сообщений shardchain. Аналогичным образом, наиболее важной неразделенной частью состояния shardchain является *OutMsgQueue, очередь исходящих сообщений. Он содержит недоставленные* сообщения, включенные в *OutMsgDescr, либо последним блоком shardchain, ведущим к этому состоянию, либо одним из его предшественников.*

Первоначально каждое исходящее сообщение включается в *OutMsgQueue; оно удаляется из очереди только после того, как оно либо было включено в InMsgDescr* блока «соседнего» шардчейна (следующего по отношению к Hypercube Routing), либо было доставлено (т.е. появилось в *InMsgDescr)* его конечного назначения через Instant Hypercube Routing. В обоих случаях *причина* удаления сообщения из *OutMsgQueue* указывается в *OutMsgDescr* блока, в котором произошло такое преобразование состояния.

1.2.8. Компоновка *InMsgDescr, OutMsgDescr* и *OutMsgQueue. Все наиболее важные неразделенные структуры данных shardchain, связанные с сообщениями, организованы в виде хэш-карт* или *словарей* (реализованных с помощью деревьев Патрисии, сериализованных в дерево ячеек, как описано в [4, 3.3]), со следующими ключами:

- Описание входящего сообщения *InMsgDescr* использует 256-битный хэш сообщения в качестве ключа.
- В описании исходящего сообщения *OutMsgDescr* в качестве ключа используется 256-битный хэш сообщения.
- В очереди исходящих сообщений *OutMsgQueue* в качестве ключа используется 352-разрядная конкатенация 32-разрядной *workchain_id назначения, первые 64 бита адреса назначения account_id и 256-разрядный хэш сообщения.*

1.2.9. Разделенная часть блока: цепочки транзакций. Разделенная часть блока shardchain состоит из хэш-карты, отображающей некоторые учетные записи, назначенные shardchain, с «виртуальными блоками accountchain» *AccountBlock, см. (3). Такой виртуальный блок accountchain состоит из последовательного списка транзакций, связанных* с этим счетом.

1.2.10. Описание транзакции. Каждая транзакция описывается в блоке экземпляром типа *Transaction*, который содержит, в частности, следующую информацию:

- Ссылка ровно на одно *входящее сообщение* (которое также должно присутствовать в *InMsgDescr)*, которое было обработано транзакцией.
- Ссылки на несколько (возможно, ноль) *исходящих сообщений* (также присутствующих в *OutMsgDescr* и, скорее всего, включенных в *OutMsgQueue), которые были сгенерированы* транзакцией.

Транзакция состоит из вызова TVM (ср. [4]) с кодом смарт-контракта, соответствующим рассматриваемой учетной записи, загруженной в виртуальную машину, и с корневой ячейкой данных смарт-контракта, загруженной в регистр виртуальной машины c4. Само входящее сообщение передается в стеке в качестве аргумента функции main() смарт-контракта вместе с некоторыми другими важными данными, такими как количество TON Grams и других определенных валют, прикрепленных к сообщению, адрес учетной записи отправителя, текущий баланс смарт-контракта и так далее.

В дополнение к информации, перечисленной выше, *экземпляр транзакции* также содержит исходное и окончательное состояния учетной записи (т. Е. Смарт-контракта), а также некоторую статистику работы TVM (потребляемый газ, цена газа, выполненные инструкции, ячейки созданы / уничтожены, код окончания виртуальной машины и т. Д.).

1.2.11. Разделенная часть состояния шардчейна: состояния счета. Напомним, что, согласно пункту (3), разделенная часть состояния shardchain состоит из хэш-карты, отображающей каждый «определенный» идентификатор учетной записи (принадлежащий рассматриваемой shardchain) состоянию соответствующей учетной записи, заданной экземпляром типа *AccountState*.

1.2.12. Состояние счета. Само состояние счета приблизительно состоит из следующих данных:

- Его *баланс* в граммах и (опционально) в некоторых других определенных криптовалютах/токенах.

- Код *смарт-контракта или хэш кода смарт-контракта, если он будет предоставлен (загружен) позже отдельным сообщением.*

- Постоянные *данные смарт-контрактов, которые могут быть пустыми для простых смарт-контрактов. Это дерево ячеек, корень которых загружается в регистр c4 при выполнении смарт-контракта.*

- Его *статистика использования хранилища, включая количество ячеек и байтов, хранящихся в постоянном хранилище смарт-контракта (то есть внутри состояния блокчейна), и последний раз, когда с этой учетной записи взималась плата за использование хранилища.*

- Необязательное *формальное описание интерфейса* (предназначенное для смарт-контрактов) и/или *общедоступная информация о пользователях* (предназначенная в основном для пользователей и организаций).

Обратите внимание, что в блокчейне TON нет различия между «умным контрактом» и «учетной записью». Вместо этого «простые» или «кошельковые» учетные записи, обычно используемые пользователями-людьми и их приложениями криптовалютного кошелька для простых криптовалютных переводов, являются просто простыми смарт-контрактами со стандартным (общим) кодом и с постоянными данными, состоящими из открытого ключа кошелька (или нескольких открытых ключей в случае кошелька с несколькими подписями; см. 1.7.6 для более подробной информации).

1.2.13. Мастерчейн блоков. В дополнение к блокам шардчейна и их состояниям, блокчейн TON содержит *блоки мастерчейна* и *состояние мастерчейна* (также называемое *глобальным состоянием). Блоки мастерчейна и состояние очень похожи на блоки шардчейна и состояние, рассмотренные до сих пор, с некоторыми заметными различиями:*

- Мастерчейн не может быть разделен или объединен, поэтому блок мастерчейна обычно имеет ровно один непосредственный

предшественник. Единственным исключением является «блок мастерчейна ноль», отличающийся наличием порядкового номера, равного нулю; он вообще не имеет предшественников и содержит начальную конфигурацию всего блокчейна TON (например, исходный набор валидаторов).

- Блоки мастерчейна содержат еще одну важную неразделенную структуру: *ShardHashes, двоичное дерево со списком всех определенных шардчейнов вместе с хэшами последнего блока внутри каждого из перечисленных цепей шардов. Именно включение блока shardchain в эту структуру делает блок shardchain «каноническим» и позволяет другим блокам shardchains ссылаться на данные (например, исходящие сообщения), содержащиеся в блоке shardchain.*

- Состояние мастерчейна содержит глобальные параметры конфигурации всего TON Blockchain, такие как минимальная и максимальная цены на газ, поддерживаемые версии TVM, минимальная ставка для кандидатов на валидатор, список альтернативных криптовалют, поддерживаемых в дополнение к Grams, общее количество Grams, выпущенных до сих пор, и текущий набор валидаторов, ответственных за создание и подписание новых блоков, вместе с их открытыми ключами.

- Состояние мастерчейна также содержит код смарт-контрактов, используемых для выбора последующих наборов валидаторов и изменения параметров глобальной конфигурации. Сам код этих смарт-контрактов является частью глобальных параметров конфигурации и может быть соответствующим образом изменен. В этом отношении этот код (наряду с текущими значениями этих параметров) функционирует как «конституция» для блокчейна TON. Первоначально он устанавливается в нулевом блоке мастерчейна.

- Транзитные сообщения через мастерчейн отсутствуют: каждое входящее сообщение должно иметь место назначения внутри

мастерчейна, а каждое исходящее сообщение должно иметь источник внутри мастерчейна.

1.3 Условия согласованности

В дополнение к структурам данных, содержащимся в блоке и в состоянии блокчейна, которые сериализуются в мешки ячеек в соответствии с определенными схемами TL-B, подробно описанными ниже (см. главы 3-5), важной составляющей макета блокчейна являются *условия согласованности* между данными, хранящимися внутри одного или в разных блоках (как упоминалось в 1.2.3). В этом разделе подробно описана функция условий согласованности в блокчейне.

1.3.1. Выражение условий согласованности. В принципе, зависимые типы данных (например, используемые в TL-B) могут использоваться не только для описания сериализации блочных данных, но и для выражения условий, налагаемых на компоненты таких типов данных. (Например, можно определить тип данных *OrderedIntPair, с парами целых чисел* (x,y), такими, что *x < y в виде значений.) Однако TL-B в настоящее время недостаточно выразителен, чтобы закодировать все необходимые нам условия согласованности, поэтому мы выбираем полуформализованный подход в этом тексте. В будущем мы можем представить последующую полную формализацию в подходящем помощнике доказательства, таком как Coq.*

1.3.2. Важность условий согласованности. Условия согласованности в конечном итоге по крайней мере так же важны, как и «неограниченные» структуры данных, на которые они налагаются, особенно в контексте блокчейна. Например, условия согласованности гарантируют, что состояние учетной записи не изменяется между блоками и что оно может изменяться в блоке только в результате транзакции. Таким образом, условия согласованности обеспечивают безопасное хранение криптовалютных балансов и другой информации внутри блокчейна.

1.3.3. Виды условий консистенции. Существует несколько видов условий согласованности, налагаемых на блокчейн TON:

- *Глобальные условия* — Выражают инварианты по всему блокчейну TON. Например, *гарантии доставки сообщений, которые утверждают, что каждое сгенерированное сообщение должно*

быть доставлено на его целевую учетную запись и доставлено ровно один раз, являются частью глобальных условий.

- *Внутренние (локальные) условия* — Выражают условия, предъявляемые к данным, хранящимся внутри одного блока. Например, каждая транзакция, входящая в блок (т.е. присутствующая в списке транзакций какого-либо аккаунта), обрабатывает ровно одно входящее сообщение; это входящее сообщение также должно быть указано в *InMsgDescr.*структуре Блока

- *Внешние (локальные) условия* — Выражают условия, налагаемые на данные разных блоков, обычно принадлежащих к одному и тому же или к соседним шардчейнам (в отношении Hypercube Routing). Поэтому внешние условия бывают нескольких вкусов:

  – *Условия антесессора/преемника* — Выражают условия, налагаемые на данные некоторого блока и его непосредственного предшественника или (в случае предшествующего события слияния шардчейна) двух непосредственных антесессоров. Наиболее важным из этих условий является то, в котором говорится, что начальное состояние блока шардчейна должно совпадать с конечным состоянием шардчейна непосредственного блока предшественника, при условии, что между ними не произошло события разделения/слияния шардчейна.

  – *Условия Masterchain/shardchain* — Выражают условия, наложенные на блок shardchain и на блок masterchain, который ссылается на него в своем *списке ShardHashes* или упоминается в заголовке блока shardchain.

  – *Соседние (блоковые) условия* — Выражают отношения между блоками соседних шардчейнов относительно Hypercube Routing. Наиболее важные из этих условий выражают отношение между *InMsgDescr* блока и *OutMsgQueue* состояния соседнего блока.

1.3.4 Разложение глобальных и локальных условий на более простые локальные условия. Условия *глобальной* согласованности, такие как

гарантии доставки сообщений, действительно необходимы для правильной работы блокчейна; однако их трудно обеспечить и проверить напрямую. Поэтому вместо этого мы вводим множество более простых *условий локальной* согласованности, которые легче применять и проверять, поскольку они включают только один блок или, возможно, два соседних блока. Эти локальные условия выбраны таким образом, что желаемые глобальные условия являются логическими следствиями (соединения) всех локальных условий. В этой связи мы говорим, что глобальные условия были «разложены» на более простые локальные условия.

Иногда локальное условие все же оказывается слишком громоздким для принудительного исполнения или проверки. В этом случае он разлагается дальше, в еще более простые локальные условия.

1.3.5 Декомпозиция может потребовать дополнительных структур данных и дополнительных внутренних условий согласованности. Декомпозиция условия на более простые условия локальной согласованности иногда требует введения дополнительных структур данных. Например, *InMsgDescr* явно перечисляет все входящие сообщения, обработанные в блоке, даже если этот список мог быть получен путем сканирования списка всех транзакций, присутствующих в блоке. Тем не менее, *InMsgDescr* значительно упрощает соседние условия, связанные с пересылкой и маршрутизацией сообщений, которые в конечном итоге складываются в глобальные гарантии доставки сообщений.

Обратите внимание, что введение таких дополнительных структур данных является своего рода «денормализацией базы данных» (т. е. это приводит к некоторой избыточности или к тому, что некоторые данные присутствуют более одного раза), и поэтому необходимо наложить больше внутренних условий согласованности (например, если некоторые данные теперь присутствуют в двух копиях, мы должны требовать, чтобы эти две копии совпадали). Например, как только мы введем *InMsgDescr* для облегчения пересылки сообщений между shardchains, нам нужно ввести внутренние условия согласованности, связывающие *InMsgDescr* со списком транзакций того же блока.

1.3.6 Правильные условия сериализации. Помимо высокоуровневых условий внутренней согласованности, которые рассматривают содержимое

блока как значение абстрактного типа данных, существуют некоторые условия внутренней согласованности более низкого уровня, называемые «(правильными) условиями сериализации», которые гарантируют, что дерево ячеек, присутствующих в блоке, действительно является допустимой сериализацией значения ожидаемого абстрактного типа данных. Такие условия сериализации могут быть автоматически сгенерированы из схемы TL-B, описывающей абстрактный тип данных и его сериализацию в дерево ячеек.

Обратите внимание, что условия сериализации представляют собой набор взаимно рекурсивных предикатов на ячейках или срезах ячеек. Например, если значение типа A состоит из 32-битного магического числа mA, *64-битного целого числа* l *и двух ссылок на ячейки, содержащие значения типов* B и C *соответственно, то правильное условие сериализации для значений типа* A потребует, чтобы ячейка или срез ячейки содержали ровно 96 бит данных и две ссылки на ячейки $r1$ и $r2$, *с дополнительными требованиями, чтобы первые 32 бита данных содержали* mA, *и две ячейки, обозначаемые* $r1$ и $r2$, удовлетворяли условиям сериализации для значений типов B и C *соответственно.*

1.3.7 Конструктивное устранение кванторов существования. Местные условия, которые иногда можно было бы навязать, *неконструируемы, а это означает, что они не обязательно содержат объяснение того, почему они истинны. Типичный пример такого условия* C приведен

$$C: \equiv \forall_{(x:\,X)}\, \exists_{(y:\,Y)}\, A_{(x,\,y)} \qquad (5)$$

"для любого x из X *существует* y из Y, такое, что условие $A(x,y)$ выполняется". Даже если мы знаем, *что* C истинен, у нас нет способа быстро найти $y : Y$, такой, что $A(x,y)$, *для данного* $x : X$. *Как следствие, проверка* C может быть довольно трудоемкой.

Чтобы упростить проверку локальных условий, их делают *конструируемыми* (т.е. проверяемыми в ограниченное время) путем добавления некоторых *структур данных-свидетелей*. Например, условие C из (5) может быть преобразовано путем добавления новой структуры данных $f : X \to Y$ (карта f от X до Y) и введения вместо *условия* C' этого следующего

$$C' :\equiv \forall_{(x:X)} A\big(x, f(x)\big) \quad . \tag{6}$$

Конечно, "свидетельское" значение *f(x) : Y* может быть включено в (модифицированный) тип данных X вместо того, чтобы храниться в отдельной таблице *f.*

1.3.8. Пример: условие согласованности для *InMsgDescr*. *Например, условие согласованности между* X := *InMsgDescr, списком всех входящих сообщений, обработанных в блоке, и* Y := *Transactions, списком всех транзакций, присутствующих в блоке, имеет следующий вид: «Для любого входного сообщения x,* присутствующего в *InMsgDescr, транзакция y* должна присутствовать в блоке таким образом, чтобы *y* обрабатывала *x».*[8] Процедура устранения ∃, описанная в 1.3.7, приводит к введению дополнительного поля в дескрипторы входящих сообщений *InMsgDescr, содержащего ссылку на транзакцию, в которой сообщение фактически обрабатывается.*

1.3.9. Конструктивное устранение логических расхождений. Аналогично преобразованию, описанному в разделе 1.3.7, условие

$$D :\equiv \forall_{(x:X)} \big(A_1(x) \vee A_2(x)\big) \quad , \tag{7}$$

"для всех *x* из *X, по меньшей мере один из* $A_1(x)$ и $A_2(x)$ содержит", может быть преобразован в функцию $i : X \to \mathbf{2} = \{1,2\}$ и новое условие

$$D' : \equiv \forall_{(x:X)} A_{i(x)}(x) \tag{8}$$

Это частный случай элиминации экзистенциального квантора, рассмотренный ранее для $Y = \mathbf{2} = \{1,2\}$. Это может быть полезно, когда $A_1(x)$ и $A_2(x)$ являются сложными условиями, которые не могут быть быстро проверены, так что полезно заранее знать, какой из них на самом деле верен.

Например, *InMsgDescr, как рассмотрено в 1.3.8, может содержать как сообщения, обработанные в блоке, так и транзитные сообщения. Мы*

[8] Этот пример немного упрощен, так как не учитывает наличие в *InMsgDescr транзитных сообщений, которые не обрабатываются какой-либо явной транзакцией.*

могли бы ввести поле в описание входящего сообщения, чтобы указать, является ли сообщение транзитным или нет, и, в последнем случае, включить поле свидетеля для транзакции, обрабатывающей сообщение.

1.3.10. Конструктивизация условий. Этот процесс устранения неконструируемых логических связующих ∃ (квантификатор существования) и (иногда) ∨ (логическая дизъюнкция) путем введения дополнительных структур данных и полей, то есть процесса создания конструктивизма условия, будет называться *конструктивизацией. Если довести этот процесс до теоретического предела, то получается логические формулы, содержащие только универсальные квантификаторы и логические соединения, за счет добавления некоторых полей-свидетелей в определенные структуры данных.*

1.3.11. Условия действия блока. В конечном счете, все внутренние условия для блока, наряду с локальным предшественником и соседними условиями, включающими этот блок и другой ранее сгенерированный блок, составляют *условия действительности* для блока шардчейна или мастерчейна. Блок действителен, если он удовлетворяет условиям действительности. Валидаторы несут ответственность за генерацию допустимых блоков, а также за проверку валидности блоков, сгенерированных другими валидаторами.

1.3.12. Свидетели недействительности блокировки. Если блок не удовлетворяет всем условиям действительности $C_1,\ldots, C_n$ (т.е. соединению $V :\equiv \bigwedge_i C_i$ условий валидности), он *является недействительным. Это означает, что он удовлетворяет «условию недействительности»* $\neg V = \bigvee_i \neg C_i$. Если все C_i — *и, следовательно, также* V — были «конструированы» в смысле, описанном в 1.3.10, так что они содержат только логические соединения и универсальные кванторы (и простые атомные предложения), то $\neg V$ содержит только логические дизъюнкции и экзистенциальные кванторы. Затем может быть определена конструктивизация $\neg V$, которая будет включать *свидетеля недействительности, начиная с индекса* i конкретного условия валидности Ci, которое не выполняется.

Такие свидетели недействительности также могут быть сериализованы и представлены другим валидаторам или помещены в мастерчейн, чтобы

доказать, что конкретный блок или кандидат на блок действительно недействителен. Таким образом, построение и сериализация свидетелей недействительности является важной частью блокчейн-дизайна Proof-of-Stake (PoS).[9]

1.3.13. Минимизация размера свидетелей. Важным соображением для проектирования локальных условий, их разложения на более простые условия и их конструктивизации является то, чтобы сделать проверку каждого условия как можно более простой. Однако другое требование состоит в том, что мы должны минимизировать размер свидетелей как для условия (чтобы размер блока не слишком увеличивался в процессе конструктивизации), так и для его отрицания (чтобы доказательства недействительности имели ограниченный размер, что упрощает их проверку, передачу и включение в мастерчейн). Эти два принципа проектирования иногда противоречат друг другу, и тогда необходимо искать компромисс.

1.3.14. Минимизация размера доказательств Меркла. Условия согласованности первоначально предназначались для обработки стороной, которая уже имеет все соответствующие данные (например, все блоки, упомянутые в условии). В некоторых случаях, однако, они должны быть проверены стороной, которая не имеет всех рассматриваемых блоков, но знает только их хэши. Например, предположим, что доказательство недействительности блока было дополнено подписью валидатора, который подписал недействительный блок (и, следовательно, должен был быть наказан). В этом случае подпись будет содержать только хэш неправильно подписанного блока; сам блок должен быть восстановлен из другого места, прежде чем проверять доказательство недействительности блока.

Компромисс между предоставлением только хэша предположительно недопустимого блока и предоставлением всего недопустимого блока вместе со свидетелем недействительности заключается в том, чтобы дополнить свидетеля недействительности доказательством Меркла, начиная с хэша блока (то есть корневой ячейки блока). Такое доказательство

[9] Интересно отметить, что эта часть работы может быть выполнена практически автоматически.

будет включать в себя все клетки, упомянутые в свидетеле недействительности, а также все клетки на путях от этих клеток к корневым клеткам и хэши их братьев и сестер. Тогда доказательство недействительности становится достаточно самодостаточным, чтобы само по себе обеспечить достаточное обоснование для наказания валидатора. Например, доказательство недействительности, предложенное выше, может быть представлено смарт-контракту, находящемуся в мастерчейне, который наказывает валидаторов за неправильное поведение.

Поскольку такое доказательство недействительности должно быть дополнено доказательством Меркла, имеет смысл записать условия непротиворечивости так, чтобы доказательства Меркла для их отрицаний были как можно меньше. В частности, каждое отдельное состояние должно быть максимально «локальным» (т.е. включать минимальное количество клеток). Это также оптимизирует время проверки доказательства недействительности.

1.3.15. Сопоставленные данные для внешних условий. Когда валидатор предлагает неподписанный блок другим валидаторам шардчейна, эти другие валидаторы должны проверить достоверность этого кандидата в блоки, т. е. убедиться, что он удовлетворяет всем внутренним и внешним условиям локальной согласованности. В то время как внутренние условия не требуют каких-либо дополнительных данных в дополнение к самому кандидату в блоки, внешние условия нуждаются в некоторых других блоках или, по крайней мере, некоторой информации из этих блоков. Такая дополнительная информация может быть извлечена из этих блоков вместе со всеми ячейками на путях из ячеек, содержащих необходимую дополнительную информацию, в корневую ячейку соответствующих блоков и хэши братьев и сестер ячеек на этих путях, чтобы представить доказательство Меркла, которое может быть обработано без знания самих упомянутых блоков.

Эта дополнительная информация, называемая собранными *данными, сериализуется в виде пакета ячеек и представляется валидатором вместе с самим кандидатом в блоки без знака. Блок-кандидат вместе с собранными данными называется собранным блоком.*

1.3.16. Условия для сопоставленного блока. Таким образом, внешние условия согласованности для кандидата в блоки (автоматически) преобразуются во *внутренние* условия согласованности для сопоставленного блока, что значительно упрощает и ускоряет их проверку другими валидаторами. Однако некоторые данные, такие как конечное состояние непосредственного предшественника проверяемого блока, не сопоставляются. Вместо этого все валидаторы должны хранить локальную копию этих данных.

1.3.17. Условия представительства и хэши представительства. Обратите внимание, что как только доказательства Меркла включены в собранный блок, условия согласованности должны учитывать, какие данные (т. Е. Какие ячейки) фактически присутствуют в собранном блоке, а не просто упоминаются их хэшами. Это приводит к новой группе условий, называемых условиями *представления, которые должны быть в состоянии отличить внешнюю ссылку на ячейку (обычно представленную ее 256-битным хэшем) от самой ячейки. Валидатор может быть наказан за предложение собранного блока, который не содержит всех ожидаемых сопоставленных данных внутри, даже если сам кандидат в блок действителен.*

Это также приводит к использованию *репрезентативных хэшей* вместо прозрачных *хэшей* для собранных блоков.

1.3.18. Верификация при отсутствии собранных данных. Обратите внимание, что блок по-прежнему должен быть проверяемым в отсутствие собранных данных; в противном случае ни одна из сторон, кроме валидаторов, не сможет проверить ранее зафиксированный блок своими собственными средствами. В частности, свидетели не могут быть включены в собранные данные: они должны находиться в самом блоке. Собранные данные должны содержать только некоторые части соседних блоков, упомянутых в основном блоке, вместе с подходящими доказательствами Меркла, которые могут быть реконструированы любым, у кого есть сами блоки, на которые имеются ссылки.

1.3.19. Включение доказательств Меркла в сам блок. Обратите внимание, что в некоторых случаях доказательства Меркла должны быть встроены в сам блок, а не только в собранные данные. Например:

- Во время мгновенной маршрутизации гиперкубов (IHR) сообщение может быть включено непосредственно в *InMsgDescr* блока шардчейна назначения, не проходя весь путь по краям гиперкуба. В этом случае доказательство Меркла о существовании сообщения в *OutMsgDescr* блока исходного шардчейна должно быть включено в *InMsgDescr* вместе с самим сообщением.

- Доказательство недействительности или другое доказательство неправомерного поведения валидатора может быть зафиксировано в мастерчейне путем включения его в тело сообщения, отправленного на специальный смарт-контракт. В этом случае доказательство недействительности должно включать некоторые ячейки вместе с доказательством Меркла, которое, следовательно, должно содержаться в теле сообщения.

- Аналогичным образом, смарт-контракт, определяющий платежный канал или другой вид боковой цепочки, может принимать сообщения о завершении или сообщения с доказательством неправильного поведения, которые содержат подходящие доказательства Меркла.

- Конечное состояние шардчейна не входит в блок шардчейна. Вместо этого включаются только те ячейки, которые были изменены; те клетки, которые унаследованы от старого состояния, называются их хэшами, наряду с подходящими доказательствами Меркла, состоящими из клеток на пути от корня старого состояния к клеткам старого состояния, о которых идет речь.

1.3.20. Положения об обработке неполных данных. Как мы видели, необходимо включать неполные данные и доказательства Меркла в тело блока, в тело некоторых сообщений, содержащихся в блоке, и в состояние. Эта необходимость отражается некоторыми дополнительными условиями представления, а также положениями о том, что сообщения (и, соответственно, клеточные деревья, обработанные TVM) содержат

неполные данные (внешние ссылки на ячейки и доказательства Меркла). В большинстве случаев такие внешние ссылки на ячейки содержат только 256-битный хэш sha256 ячейки вместе с флагом; если смарт-контракт пытается проверить содержимое такой ячейки примитивом CTOS (например, для десериализации), срабатывает исключение. Однако внешняя ссылка на такую ячейку может быть сохранена в постоянном хранилище смарт-контракта, и могут быть вычислены как прозрачные, так и репрезентативные хэши такой ячейки.

1.4 Логическое время и логические временные интервалы

В этом разделе более подробно рассматривается так называемое логическое *время, широко используемое в блокчейне TON для пересылки сообщений и гарантий доставки сообщений, среди других целей.*

1.4.1. Логическое время. Компонентом TON Blockchain, который также играет важную роль в доставке сообщений, является *логическое время, обычно обозначаемое* Lt*. Это неотрицательное 64-битное целое число, назначенное определенным событиям примерно следующим образом:*

> Если событие e логически зависит от событий $e_1,\ldots, e_n$, *то* $Lt(e)$ является наименьшим неотрицательным целым числом, большим, чем все $Lt(e_i)$.

В частности, если $n = 0$ (т.е. если e не зависит от каких-либо предшествующих событий), то Lt(e) = 0.

1.4.2. Расслабленный вариант логического времени. В некоторых случаях мы ослабляем определение логического времени, требуя только того, чтобы

> Lt(e) > Lt(e')всякий раз, когда $e \succ e'$ $(\text{i.e.}, e$ логически зависит от *e'), (9)*

не настаивая на том, чтобы Lt(e) был наименьшим неотрицательным целым числом с этим свойством. В таких случаях можно говорить о *расслабленном* логическом времени, в отличие от строгого логического времени, определенного выше (см. 1.4.1). Заметьте, однако, что условие (9) является фундаментальным свойством логического времени и не может быть ослаблено дальше.

1.4.3. Логические временные интервалы. Имеет смысл присвоить некоторым событиям или коллекциям событий C *интервал* логических времен Lt•(C) = [Lt−(C),Lt+(C)), означающий, что сбор событий C происходил в указанном «интервале» логических времен, где Lt−(C) < Lt+(C) — некоторые целые числа (на практике 64-битные целые числа). В этом случае можно сказать, что C *начинается* в логическом времени Lt−(C) и *заканчивается* в логическом времени Lt+(C).

По умолчанию мы предполагаем, что Lt+(e) = Lt(e)+1 и Lt−(e) = Lt(e) для простых или «атомных» событий, предполагая, что они длятся ровно одну единицу логического времени.

В общем, если у нас есть одно значение Lt(C), а также логический временной интервал Lt•(C) = [Lt−(C),Lt+(C)), мы всегда требуем, чтобы

$$Lt(C) \in [Lt-(C),Lt+(C)) \tag{10}$$

или, эквивалентно,

$$Lt-(C) \le Lt(C) < Lt+(C) \tag{11}$$

В большинстве случаев мы выбираем Lt(C) = Lt−(C).

1.4.4. Требования к логическим временным интервалам. Тремя основными требованиями к логическим временным интервалам являются:

- $0 \le Lt-(C) < Lt+(C)$ являются неотрицательными целыми числами для любой коллекции событий C.
- Если $e' \prec e$ (т.е. если атомное событие e логически зависит от другого атомарного события e'), *то* $Lt•(e') < Lt•(e)$ (т.е. $Lt+(e') \le Lt-(e)$).
- Если $C \supset D$ (т.е. если коллекция событий C содержит другую коллекцию событий D), *то* $Lt•(C) \supset Lt•(D)$, т.е.

$$Lt-(C) \le Lt-(D) < Lt+(D) \le Lt+(C) \tag{12}$$

  В частности, если C состоит из атомных событий $e_1,\ldots, e_n$, *то* $Lt-(C) \le \inf_i Lt-(e_i) \le \inf_i Lt(e_i)$ и $Lt+(C) \ge \sup_i Lt+(e_i) \ge 1+\sup_i Lt(e_i)$.

1.4.5. Строгие, или минимальные, логические временные интервалы. Любой конечной совокупности атомарных событий $E = \{e\}$, связанных причинно-следственным отношением (частичным порядком) можно присвоить $\prec$, а всем подмножествам $C \subset E$ — минимальные логические промежутки времени. То есть среди всех присвоений логических временных интервалов, удовлетворяющих условиям, перечисленным в 1.4.4, мы выбираем тот, который имеет все Lt+(C) − Lt−(C) как можно

меньше, а если существует несколько назначений с этим свойством, то выбираем и ту, которая имеет минимальный Lt−(C).

Такое присвоение может быть достигнуто путем сначала присвоения логического времени Lt(e) всем атомарным событиям $e \in E$, как описано в 1.4.1, а затем установки Lt−(C) := $\inf_{e \in C}$ Lt(e) и Lt+(C) := 1 + $\sup_{e \in C}$ Lt(e) для любого $C \subset E$.

В большинстве случаев, когда нам нужно назначить логические временные интервалы, мы используем минимальные логические временные интервалы, только что описанные.

1.4.6. Логическое время в блокчейне TON. Блокчейн TON присваивает логическое время и логические временные интервалы нескольким своим компонентам.

Например, каждому исходящему сообщению, созданному в транзакции, присваивается *время логического создания; для этого создание исходящего сообщения считается атомарным событием, логически зависящим от предыдущего сообщения, созданного той же транзакцией, а также от предыдущей транзакции той же учетной записи, от входящего сообщения, обработанного той же транзакцией, и на всех событиях, содержащихся в блоках, на которые ссылаются хэши, содержащиеся в блоке с той же транзакцией. Как следствие, исходящие сообщения, созданные одним и тем же смарт-контрактом, строго увеличивают время логического создания.* Сама транзакция считается совокупностью атомарных событий, и ей присваивается логический временной интервал (см. 4.2.1 для более точного описания).

Каждый блок представляет собой коллекцию событий создания транзакций и сообщений, поэтому ему присваивается логический интервал времени, явно указанный в заголовке блока.

1.5 Общее состояние блокчейна

В этом разделе обсуждается общее состояние блокчейна TON, а также состояния отдельных шардчейнов и мастерчейнов. Например, точное определение состояния соседних шардчейнов становится решающим для правильной формализации условия непротиворечивости, утверждающего, что валидаторы для шардчейна должны импортировать самые старые сообщения из союза *OutMsgQueues, взятые из состояний всех соседних шардчейнов (см. 2.2.5).*

1.5.1. Общее состояние, определяемое блоком мастерчейна. Каждый блок мастерчейна содержит список всех активных в данный момент сегментов и последних блоков для каждого из них. В этом отношении *каждый блок мастерчейна определяет соответствующее общее состояние блокчейна TON, поскольку он фиксирует состояние каждого шардчейна, а также мастерчейна.*

Важное требование, предъявляемое к этому списку последних блоков для всех блоков шардчейна, заключается в том, что, если блок мастерчейна B перечисляет S как последний блок некоторого шардчейна, а более новый блок мастерчейна B', *с* B в качестве одного из его предшественников, перечисляет S' как последний блок того же шардчейна, то S должен быть одним из предшественников S'.[10] Это условие делает общее состояние блокчейн TON определяется последующим блоком мастерчейна B', совместимым с общим состоянием, определенным предыдущим блоком B.

1.5.2. Общее состояние, определяемое блоком шардчейна. Каждый блок shardchain содержит хэш последнего блока masterchain в своем заголовке. Следовательно, все блоки, упомянутые в этом блоке мастерчейна, вместе с их предшественниками, считаются «известными» или «видимыми» для блока шардчейна, и никакие другие блоки не видны ему, за исключением его предшественников внутри его собственного шардчейна.

В частности, когда мы говорим, что блок *должен* импортировать в свой *InMsgDescr* сообщения из *OutMsgQueue* состояний всех соседних шардчейнов, это означает, что именно блоки других шардчейнов, видимые этому блоку, должны быть приняты во внимание, и в то же время блок не может содержать сообщения из «невидимых» блоков, даже если в остальном они верны.

[10] Чтобы правильно выразить это условие при наличии динамического шардинга, следует зафиксировать некоторый счет ξ, *и рассмотреть последние блоки* S и $S0$ шардчейнов, содержащих ξ в конфигурациях осколок как B, так и $B0$, *так как осколки, содержащие* ξ могут отличаться в B и B'

1.6 Настраиваемые параметры и смарт-контракты

Напомним, что TON Blockchain имеет несколько так называемых «настраиваемых параметров» (ср. [3]), которые являются либо определенными значениями, либо определенными смарт-контрактами, находящимися в мастерчейне. В этом разделе описывается хранение и доступ к этим настраиваемым параметрам.

1.6.1. Примеры настраиваемых параметров. К свойствам блокчейна, управляемым настраиваемыми параметрами, относятся:

- Минимальная ставка для валидаторов.
- Максимальный размер группы избранных валидаторов.
- Максимальное количество блоков, за которые отвечает одна и та же группа валидаторов.
- Процесс выборов валидатора.
- Процесс наказания валидатора.
- Текущий активный и следующий избранный набор валидаторов.
- Процесс изменения настраиваемых параметров и адрес смарт-контракта γ ответственным за хранение значений настраиваемых параметров и за изменение их значений.

1.6.2. Расположение значений настраиваемых параметров. Настраиваемые параметры хранятся в постоянных данных специальной конфигурации смарт-контракта γ, находящейся в мастерчейне блокчейна TON. Точнее, первой ссылкой на корневую ячейку постоянных данных этого смарт-контракта является словарь, отображающий 64-битные ключи (номера параметров) со значениями соответствующих параметров; каждое значение сериализуется в срез ячейки в соответствии с типом этого значения. Если значение является «смарт-контрактом» (обязательно находящимся в мастерчейне), вместо него используется 256-битный адрес учетной записи.

1.6.3. Быстрый доступ через заголовок мастерчейн блоков. Чтобы упростить доступ к текущим значениям настраиваемых параметров, и сократить доказательства Меркла, содержащие ссылки на них, заголовок каждого блока мастерчейна содержит адрес смарт-контракта *γ. Он также содержит прямую ячеистую ссылку на словарь, содержащий все значения настраиваемых параметров, которая лежит в постоянных данных γ. Дополнительные условия согласованности гарантируют, что эта ссылка совпадает с той, которая получена путем проверки конечного состояния смарт-контракта γ.*

1.6.4. Получение значений настраиваемых параметров методами get. Конфигурация смарт-контракта *γ* предоставляет доступ к некоторым настраиваемым параметрам с помощью «методов get». Эти специальные методы смарт-контракта не изменяют его состояние, а вместо этого возвращают необходимые данные в стек TVM.

1.6.5. Получение значений настраиваемых параметров путем получения сообщений. Аналогичным образом, конфигурация смарт-контракта *γ* может определять некоторые «обычные» методы (т.е. специальные входящие сообщения) для запроса значений определенных параметров конфигурации, которые будут отправляться в исходящих сообщениях, генерируемых транзакцией, обрабатывающей такое входящее сообщение. Это может быть полезно для некоторых других фундаментальных смарт-контрактов, которые должны знать значения определенных параметров конфигурации.

1.6.6. Значения, полученные методами get, могут отличаться от значений, полученных через заголовок блока. Обратите внимание, что состояние γ смарт-контракта конфигурации, *включая значения настраиваемых параметров, может изменяться несколько раз внутри блока мастерчейна, если в этом блоке несколько транзакций, обработанных γ.* Как следствие, значения, полученные путем вызова методов get *γ или отправки get сообщений γ, могут отличаться от значений, полученных при проверке ссылки в заголовке блока (см. 1.6.3), которая относится к конечному* состоянию настраиваемых параметров в блоке.

1.6.7. Изменение значений настраиваемых параметров. Процедура изменения значений настраиваемых параметров определена в коде смарт-контракта γ. *Для большинства настраиваемых параметров, называемых обычными, любой валидатор может предложить новое значение, отправив* γ *специальное сообщение с номером параметра и его предлагаемым значением. Если предложенное значение является допустимым, смарт-контракт собирает дальнейшие сообщения о голосовании от валидаторов, и если более двух третей каждого из текущего и следующего наборов валидаторов поддерживают предложение, значение изменяется.*

Некоторые параметры, такие как текущий набор валидаторов, не могут быть изменены таким образом. Вместо этого текущая конфигурация содержит параметр с адресом смарт-контракта ν, ответственным за выбор следующего набора валидаторов, а смарт-контракт γ принимает сообщения только от этого смарт-контракта ν для изменения значения параметра конфигурации, содержащего текущий набор валидаторов.

1.6.8. Изменение процедуры выборов валидатора. Если процедура выбора валидатора когда-либо должна быть изменена, это может быть достигнуто путем сначала фиксации нового смарт-контракта выбора валидатора в мастерчейне, а затем изменения обычного настраиваемого параметра, содержащего адрес ν смарт-контракта выбора валидатора. Для этого потребуется, чтобы две трети валидаторов приняли предложение в ходе голосования, как описано выше в разделе 1.6.7.

1.6.9. Изменение процедуры изменения настраиваемых параметров. Аналогичным образом, адрес самого смарт-контракта конфигурации является настраиваемым параметром и может быть изменен таким образом. Таким образом, большинство фундаментальных параметров и смарт-контрактов TON Blockchain могут быть изменены в любом направлении, согласованном квалифицированным большинством валидаторов.

1.6.10. Начальные значения настраиваемых параметров. Начальные значения большинства настраиваемых параметров отображаются в нулевом блоке мастерчейна как часть начального состояния мастерчейна,

которое явно присутствует без упущения в этом блоке. Код всех фундаментальных смарт-контрактов также присутствует в исходном состоянии. Таким образом, оригинальная «конституция» и конфигурация блокчейна TON, включая оригинальный набор валидаторов, становятся явными в нулевом блоке.

1.7 Новые смарт-контракты и их адреса

В этом разделе обсуждается создание и инициализация новых смарт-контрактов, в частности, происхождение их исходного кода, постоянные данные и баланс. В нем также обсуждается назначение адресов учетных записей новым смарт-контрактам.

1.7.1. Описание действительно только для мастерчейна и базового рабочего цепя. Механизмы создания новых смарт-контрактов и назначения их адресов, описанные в этом разделе, действительны только для базовой рабочей цепи и мастерчейна. Другие рабочие цепи могут определять свои собственные механизмы решения этих проблем.

1.7.2. Перевод криптовалюты на неинициализированные счета. Прежде всего, *можно отправлять сообщения, в том числе значимые, на ранее не упоминавшиеся аккаунты.* Если входящее сообщение поступает в shardchain с адресом назначения η, соответствующим неопределенной учетной записи, оно обрабатывается транзакцией, как если бы код смарт-контракта был пустым (т. Е. Состоящим из неявного RET). Если сообщение имеет ценность, это приводит к созданию «неинициализированного счета», который может иметь ненулевой баланс (если на него были отправлены сообщения, содержащие ценность), [11] но не имеет кода и данных. Поскольку даже неинициализированная учетная запись занимает некоторое постоянное хранилище (необходимое для хранения своего баланса), некоторые небольшие платежи по постоянному хранению будут время от времени взиматься с баланса счета, пока он не станет отрицательным.

1.7.3. Инициализация смарт-контрактов по сообщениям конструктора. Учетная запись, или смарт-контракт, создается путем отправки специального *сообщения конструктора* M на его адрес η. *Тело такого сообщения содержит дерево ячеек с начальным кодом смарт-контракта (который в некоторых ситуациях может быть заменен его хэшем), и* исходные данные смарт-контракта (возможно, пустые; его можно

[11] Значимые сообщения с установленным флагом bounce не будут приниматься неинициализированной учетной записью, а будут «возвращены» обратно.

заменить его хэшем). Хэш кода и данных, содержащихся в сообщении конструктора, должен совпадать с адресом η смарт-контракта; в противном случае он отклоняется.

После инициализации кода и данных смарт-контракта из тела сообщения конструктора остальная часть сообщения конструктора обрабатывается транзакцией (*транзакцией создания* для смарт-контракта η*) путем вызова TVM способом, аналогичным тому, который используется для обработки обычных входящих сообщений.*

1.7.4. Начальный баланс смарт-контракта. Обратите внимание, что сообщение конструктора обычно должно содержать некоторое значение, которое будет передано на баланс вновь созданного смарт-контракта; в противном случае новый смарт-контракт будет иметь нулевой баланс и не сможет платить за хранение своего кода и данных в блокчейне. Минимальный баланс, требуемый от вновь созданного смарт-контракта, является линейной (точнее, аффинной) функцией хранилища, которое он использует. Коэффициенты этой функции могут зависеть от рабочей цепи; в частности, они выше в мастерчейне, чем в основной рабочей цепи.

1.7.5. Создание смарт-контрактов внешними сообщениями конструктора. В некоторых случаях необходимо создать смарт-контракт с помощью сообщения конструктора, которое не может иметь никакого значения, например, с помощью сообщения конструктора «из ниоткуда» (внешнего входящего сообщения). Затем следует сначала перевести достаточное количество средств на неинициализированный смарт-контракт, как описано в 1.7.2, и только потом отправлять конструктору сообщение «из ниоткуда».

1.7.6. Пример: создание смарт-контракта криптовалютного кошелька. Примером вышеописанной ситуации являются приложения криптовалютного кошелька для пользователей-людей, которые должны создать специальный смарт-контракт кошелька в блокчейне, в котором будут храниться средства пользователя. Это может быть достигнуто следующим образом:

- Приложение криптовалютного кошелька генерирует новую криптографическую пару открытого / закрытого ключей (обычно для

криптографии эллиптической кривой Ed25519, поддерживаемой специальными примитивами TVM) для подписания будущих транзакций пользователя.

- Приложение криптовалютного кошелька знает код создаваемого смарт-контракта (который обычно одинаков для всех пользователей), а также данные, которые обычно состоят из открытого ключа кошелька (или его хэша) и генерируются в самом начале. Хэш этой информации — это адрес ξ создаваемого смарт-контракта кошелька.

- Приложение кошелька может отображать адрес пользователя ξ, *и пользователь может начать получать средства на свой неинициализированный счет* ξ, *например, купив криптовалюту на бирже или попросив друга перевести небольшую сумму.*

- Приложение кошелька может проверить shardchain, содержащий учетную запись ξ (в случае базовой учетной записи workchain) или masterchain (в случае учетной записи masterchain), либо самостоятельно, либо с помощью проводника блокчейна, и проверить баланс ξ.

- Если баланса достаточно, приложение кошелька может создать и подписать (закрытым ключом пользователя) сообщение конструктора («из ниоткуда») и отправить его для включения в валидаторы или коллаторы для соответствующего блокчейна.

- Как только сообщение конструктора включено в блок блокчейна и обработано транзакцией, смарт-контракт кошелька наконец создается.

- Когда пользователь хочет перевести некоторые средства какому-либо другому пользователю или смарт-контракту, η, *или хочет отправить ценностное сообщение* η, *он использует свое приложение кошелька для создания сообщения* m, *которое* она хочет, чтобы ее смарт-контракт кошелька ξ отправить η, *конверт* m в специальное «сообщение из ниоткуда» m' с назначением ξ, *и подписать* m' своим

закрытым ключом. Некоторые положения против повторных атак должны быть сделаны, как описано в 2.2.1.

- Смарт-контракт кошелька получает сообщение m' и проверяет действительность подписи с помощью открытого ключа, хранящегося в его постоянных данных. Если подпись верна, она извлекает встроенное сообщение m из m' и отправляет его по назначению η *с указанной суммой средств.*

- Если пользователю не нужно сразу начинать перевод средств, а только хочет пассивно получать какие-то средства, он может держать свой аккаунт неинициализированным столько, сколько захочет (при условии, что постоянные платежи за хранилище не приводят к исчерпанию его баланса), тем самым минимизируя профиль хранения и постоянные платежи за хранение учетной записи.

- Обратите внимание, что приложение кошелька может создать для пользователя-человека иллюзию, что средства хранятся в самом приложении, и предоставить интерфейс для перевода средств или отправки произвольных сообщений «напрямую» со счета пользователя ξ. *В реальности все эти операции будет выполняться смарт-контрактом кошелька пользователя, который эффективно выступает в качестве прокси для таких запросов. Мы видим, что криптовалютный кошелек является простым примером смешанного* приложения, имеющего часть on-chain (смарт-контракт кошелька, используемый в качестве прокси для исходящих сообщений) и часть offchain (приложение внешнего кошелька, работающее на устройстве пользователя и хранящее закрытый ключ учетной записи).

Конечно, это всего лишь один из способов борьбы с простейшими смарт-контрактами кошелька пользователя. Можно создать смарт-контракты кошелька с несколькими подписями или создать общий кошелек с внутренними балансами, хранящимися внутри него для каждого из его отдельных пользователей, и так далее.

1.7.7. Смарт-контракты могут быть созданы другими смарт-контрактами. Обратите внимание, что смарт-контракт может генерировать и отправлять сообщение конструктора при обработке любой транзакции. Таким образом, смарт-контракты могут автоматически создавать новые смарт-контракты, если это необходимо, без какого-либо вмешательства человека.

1.7.8. Смарт-контракты могут быть созданы смарт-контрактами кошелька. С другой стороны, пользователь может скомпилировать код для своего нового смарт-контракта ν, *сгенерировать соответствующее сообщение конструктора* m *и использовать приложение кошелька, чтобы заставить свой смарт-контракт кошелька* ξ отправить сообщение m в ν с достаточным количеством средств, тем самым создавая новый смарт-контракт ν.

1.8 Модификация и удаление смарт-контрактов

В этом разделе объясняется, как код и состояние смарт-контракта могут быть изменены, а также как и когда смарт-контракт может быть уничтожен.

1.8.1. Изменение данных смарт-контракта. Постоянные данные смарт-контракта обычно изменяются в результате выполнения кода смарт-контракта в TVM при обработке транзакции, вызванной входящим сообщением в смарт-контракт. Более конкретно, код смарт-контракта имеет доступ к старому постоянному хранилищу смарт-контракта через регистр управления TVM c4 и может модифицировать постоянное хранилище, сохраняя другое значение в c4 до обычного прекращения.

Как правило, нет других способов изменить данные существующего смарт-контракта. Если код смарт-контракта не предоставляет никаких способов изменения постоянных данных (например, если это простой смарт-контракт кошелька, как описано в 1.7.6, который инициализирует постоянные данные открытым ключом пользователя и не намерен когда-либо изменять их), то он будет фактически неизменяемым, если только код смарт-контракта не будет изменен первым.

1.8.2. Изменение кода смарт-контракта. Аналогичным образом, код существующего смарт-контракта может быть изменен только в том случае,

если некоторые положения для такого обновления присутствуют в текущем коде. Код изменяется путем вызова примитива TVM SETCODE, который устанавливает корень кода для текущего смарт-контракта из верхнего значения в стеке TVM. Изменение применяется только после обычного прекращения текущей транзакции.

Как правило, если разработчик смарт-контракта хочет иметь возможность обновить свой код в будущем, он предоставляет специальный «метод обновления кода» в исходном коде смарт-контракта, который вызывает SETCODE в ответ на определенные входящие сообщения «обновления кода», используя новый код, отправленный в самом сообщении в качестве аргумента SETCODE. Некоторые положения должны быть сделаны для защиты смарт-контракта от несанкционированной замены кода; в противном случае контроль над смарт-контрактом и средствами на его балансе может быть потерян. Например, сообщения об обновлении кода могут приниматься только с надежного исходного адреса или могут быть защищены путем требования действительной криптографической подписи и правильного порядкового номера.

1.8.3. Хранение кода или данных смарт-контракта вне блокчейна. Код или данные смарт-контракта могут храниться вне блокчейна и быть представлены только их хэшами. В таких случаях могут обрабатываться только пустые входящие сообщения, а также сообщения, содержащие правильную копию кода смарт-контракта (или его часть, относящуюся к обработке конкретного сообщения) и его данные внутри специальных полей. Примером такой ситуации служат неинициализированные смарт-контракты и сообщения конструктора, описанные в 1.7.

1.8.4. Использование библиотек кода. Некоторые смарт-контракты могут использовать один и тот же код, но использовать разные данные. Одним из примеров этого являются смарт-контракты кошельков (см. 1.7.6), которые, вероятно, будут использовать один и тот же код (во всех кошельках, созданных одним и тем же программным обеспечением), но с разными данными (потому что каждый кошелек должен использовать свою собственную пару криптографических ключей). В этом случае код для всех смарт-контрактов кошелька лучше всего фиксировать разработчиком в общей *библиотеке; эта библиотека будет находиться в мастерчейне и*

ссылаться на ее хэш с использованием специальной «ссылки на ячейку внешней библиотеки» в качестве корня кода каждого смарт-контракта кошелька (или в качестве поддерева внутри этого кода).

Обратите внимание, что даже если код библиотеки становится недоступным — например, потому, что его разработчик перестает платить за его хранение в мастерчейне — все равно можно использовать смарт-контракты, относящиеся к этой библиотеке, либо снова зафиксировав библиотеку в мастерчейне, либо включив соответствующие части в сообщение, отправленное смарт-контракту. Этот механизм разрешения внешних ячеек более подробно рассматривается далее в разделе 4.4.3.

1.8.5. Уничтожение смарт-контрактов. Обратите внимание, что смарт-контракт не может быть действительно уничтожен, пока его баланс не станет нулевым или отрицательным. Он может стать отрицательным в результате сбора постоянных платежей за хранилище или после отправки исходящего сообщения, несущего ценность, передающего почти весь свой предыдущий баланс.

Например, пользователь может решить перевести все оставшиеся средства со своего кошелька на другой кошелек или смарт-контракт. Это может быть полезно, например, если кто-то хочет обновить кошелек, но смарт-контракт кошелька не имеет никаких положений для будущих обновлений; тогда можно просто создать новый кошелек и перевести на него все средства.

1.8.6. Замороженные счета. Когда баланс счета становится неположительным после транзакции или меньше определенного минимума, зависящего от рабочей цепи, счет *замораживается* путем замены всего его кода и данных одним 32-байтовым хэшем. Этот хэш хранится впоследствии в течение некоторого времени (например, пару месяцев), чтобы предотвратить воссоздание смарт-контракта путем его первоначальной транзакции создания (которая по-прежнему имеет правильный хэш, равный адресу учетной записи), и позволить его владельцу воссоздать учетную запись, переведя некоторые средства и отправив сообщение, содержащее код и данные учетной записи, быть восстановленным в блокчейне. В этом отношении замороженные счета аналогичны неинициализированным счетам; однако хэш правильного кода

и данных для замороженного счета не обязательно равен адресу счета, а хранится отдельно.

Обратите внимание, что замороженные счета могут иметь отрицательное сальдо, что указывает на необходимость постоянных платежей по хранению. Счет не может быть разморожен до тех пор, пока его баланс не станет положительным и не превысит установленное минимальное значение.

2 Гарантии пересылки и доставки сообщений

В этой главе обсуждается пересылка сообщений внутри блокчейна TON, включая протоколы Hypercube Routing (HR) и Instant Hypercube Routing (IHR). В нем также описываются положения, необходимые для реализации гарантий доставки сообщений и гарантии заказа FIFO.

2.1 Адреса сообщений и вычисления следующего прыжка

В этом разделе объясняется вычисление транзитных адресов и адресов следующего прыжка по варианту алгоритма маршрутизации гиперкубов, используемого в TON Blockchain. Сам протокол маршрутизации гиперкубов, который использует концепции и алгоритм вычисления адресов следующего прыжка, представленные в этом разделе, представлен в следующем разделе.

2.1.1. Адреса учетных записей. Адрес *источника* и *адрес назначения* всегда присутствуют в любом сообщении. Как правило, это *(полные) адреса учетных записей. Полный адрес учетной записи состоит из workchain_id* (подписанное 32-разрядное целое число с большим порядком байтов, определяющее рабочую цепочку), за которым следует (обычно) 256-разрядный *внутренний адрес* или *идентификатор учетной записи account_id* (который также может быть интерпретирован как целое число без знака с большим порядком байтов), определяющий учетную запись в выбранной рабочей цепочке.

Различные рабочие цепи могут использовать идентификаторы учетных записей, которые короче или длиннее, чем «стандартные» 256 бит, используемые в мастерчейне (workchain_id = -1) и в базовой рабочей цепочке (workchain_id = 0). С этой целью состояние мастерчейн содержит список всех рабочих цепей, определенных до сих пор, а также длину идентификатора их учетной записи. Важным ограничением является то, что *account_id* для любой рабочей цепи, должна быть длиной не менее 64 бит.

В дальнейшем мы часто рассматриваем только случай 256-битных адресов учетных записей для простоты. Только первые 64 бита *account_id*

актуальны для целей маршрутизации сообщений и разделения шардчейнов.

2.1.2. Адреса источника и назначения сообщения. Любое сообщение имеет как исходный *адрес, так* и *адрес назначения. Его исходным адресом является адрес учетной записи (смарт-контракта), которая создала сообщение при обработке какой-либо транзакции; адрес источника не может быть изменен или задан произвольно, и смарт-контракты в значительной степени зависят от этого свойства. Напротив, при создании сообщения* может быть выбран любой правильно сформированный адрес назначения; после этого адрес назначения не может быть изменен.

2.1.3. Внешние сообщения без адреса источника или назначения. Некоторые сообщения могут не иметь адреса источника или адреса назначения (хотя хотя один из них должен присутствовать), о чем свидетельствуют специальные флаги в заголовке сообщения. Такие сообщения — это *внешние сообщения,* предназначенные для взаимодействия блокчейна TON с внешним миром — пользователями-людьми и их приложениями для криптокошельков, внецепочными и смешанными приложениями и сервисами, другими блокчейнами и так далее.

Внешние сообщения никогда не маршрутизируются внутри блокчейна TON. Вместо этого «сообщения из ниоткуда» (т.е. без адреса источника) напрямую включаются в *InMsgDescr* целевого блока shardchain (при условии соблюдения некоторых условий) и обрабатываются транзакцией в этом самом блоке. Аналогичным образом, «сообщения в никуда» (то есть без адреса назначения TON Blockchain), также известные как *сообщения журнала, также присутствуют только в блоке, содержащем транзакцию, которая сгенерировала такое сообщение*.[12]

[12] «Сообщения в никуда» могут иметь некоторые специальные поля в своем теле, указывающие на их назначение за пределами блокчейна TON — например, учетная запись в каком-либо другом блокчейне или IP-адрес и порт — которые могут быть соответствующим образом интерпретированы сторонним программным обеспечением. Такие поля игнорируются блокчейном TON.

Поэтому внешние сообщения практически не имеют отношения к обсуждению маршрутизации сообщений и гарантий доставки сообщений. Фактически, гарантии доставки сообщений для исходящих внешних сообщений тривиальны (в лучшем случае сообщение должно быть включено в *часть блока LogMsg*), а для входящих внешних сообщений их нет, поскольку валидаторы блока shardchain могут свободно включать или игнорировать предлагаемые входящие внешние сообщения по своему усмотрению (например, в соответствии с платой за обработку, предлагаемой сообщением).[13]

В дальнейшем мы сосредоточимся на «обычных» или «внутренних» сообщениях, которые имеют как источник, так и адрес назначения.

2.1.4. Транзитные и последующие адреса. Когда сообщение должно быть перенаправлено через промежуточные цепочки сегментов до достижения предполагаемого пункта назначения, ему назначается *транзитный адрес и адрес* следующего *прыжка* в дополнение к (неизменяемым) адресам источника и назначения. Когда копия сообщения находится внутри транзитного шардчейна, ожидающего его ретрансляции до следующего прыжка, транзитный *адрес* является его промежуточным адресом, лежащим в транзитном шардчейне, как будто принадлежащим специальному смарт-контракту ретрансляции сообщений, единственной задачей которого является передача неизмененного сообщения следующему шардчейну на маршруте. Адрес *следующего прыжка* — это адрес в соседнем шардчейне (или, в некоторых редких случаях, в той же шардчейне), на который необходимо передать сообщение. После ретрансляции сообщения адрес следующего прыжка обычно становится транзитным адресом копии сообщения, включенной в следующую цепочку.

[13] Проблема обхода возможной цензуры валидатора, которая может произойти, например, если все валидаторы вступают в сговор с целью не включать внешние сообщения, отправленные на учетные записи, принадлежащие к некоторому набору учетных записей, занесенных в черный список, рассматривается отдельно в другом месте. Основная идея заключается в том, что валидаторы могут быть вынуждены пообещать включить сообщение с известным хэшем в будущий блок, ничего не зная об идентификации отправителя или получателя; они должны будут сдержать это обещание впоследствии, когда будет представлено само сообщение с заранее согласованным хэшем.

Сразу после создания исходящего сообщения в цепочке shardchain (или в мастерчейне) его транзитный адрес устанавливается на исходный адрес.[14]

2.1.5. Вычисление адреса следующего прыжка для маршрутизации гиперкубов. Блокчейн TON использует вариант маршрутизации гиперкубов. Это означает, что адрес следующего прыжка вычисляется из транзитного адреса (первоначально равного исходному адресу) следующим образом:

А. с подписью биг-эндиана) разбиваются на группы *32-битные workchain_id* компоненты как транзитного адреса, так и адреса назначения (*из* n_1 бит (в настоящее время $n_1 = 32$), и они сканируются слева (т. Е. Наиболее значимые биты) справа. Если одна из групп в транзитном адресе отличается от соответствующей группы в адресе назначения, то значение этой группы в транзитном адресе заменяется ее значением в адресе назначения для вычисления адреса следующего прыжка.

Б. Если *workchain_id* части транзитного и целевого адресов совпадают, то аналогичный процесс применяется к *account_id* частям адресов: *account_id* части, а точнее их первые (наиболее значимые) 64 бита, разбиваются на группы n_2 бит (в настоящее время $n_2 = 4$ используются битовые группы, соответствующие шестнадцатеричным цифрам адреса), начиная с наиболее значимого бита, и сравниваются, начиная с левого. Первая группа, которая отличается, заменяется в транзитном адресе ее значением в адресе назначения для вычисления адреса следующего прыжка.

В. Если первые 64 бита *account_id* частей транзитного и целевого адресов также совпадают, то учетная запись назначения принадлежит текущему shardchain, и сообщение вообще не должно пересылаться за пределы текущей shardchain. Вместо этого он должен быть обработан транзакцией внутри него.

[14] Однако внутренний процесс маршрутизации, описанный в 2.1.11, применяется сразу после этого, что может дополнительно изменить транзитный адрес.

2.1.6. Обозначение адреса следующего прыжка. Обозначаем

$$\mathrm{NextHop}(\xi,\eta) \tag{13}$$

адрес следующего прыжка, вычисляемый для текущего (исходного или транзитного) адреса ξ и адреса назначения η.

2.1.7. Поддержка адресов anycast. «Большие» смарт-контракты, которые могут иметь отдельные экземпляры в разных шардчейнах, могут быть достигнуты с использованием *адресов назначения anycast. Эти адреса поддерживаются следующим образом.*

Адрес anycast (η,d) состоит из обычного адреса η вместе с его «глубиной разделения» $d \le 31$. Идея состоит в том, что сообщение может быть доставлено на любой адрес, отличающийся от η только в первых d битах части внутреннего адреса (т.е. без учета идентификатора рабочей цепи, который должен точно совпадать). Это достигается следующим образом:

- Эффективный адрес назначения $\tilde{\eta}$ вычисляется из (η,d) путем замены первых d битов внутренней адресной части η соответствующими битами, взятыми из исходного адреса ξ.

- Все вычисления NextHop(ν,η) заменяются на NextHop(ν,$\tilde{\eta}$), для $\nu = \xi$, а также для всех других промежуточных адресов ν. *Таким образом, Hypercube Routing или Instant Hypercube Routing в конечном итоге доставит сообщение в shardchain, содержащий* $\tilde{\eta}$.

- Когда сообщение обрабатывается в целевой shardchain (содержащей адрес $\tilde{\eta}$*), оно может быть обработано учетной записью* $\eta 0$ того же shardchain, отличающегося от η и $\tilde{\eta}$ только в первых d битах внутренней адресной части. Точнее, если общий префикс адреса сегмента равен s, *так что только внутренние адреса, начинающиеся с двоичной строки* s, принадлежат целевому сегменту, то η' вычисляется из η путем замены первого $\min(\mathrm{d},|\mathrm{s}|)$ биты внутренней адресной части η с соответствующими битами s.

Тем не менее, мы молчаливо игнорируем существование адресов anycast и дополнительную обработку, которую они требуют в следующих обсуждениях.

2.1.8. Хэмминговская оптимальность алгоритма адреса следующего прыжка. Обратите внимание, что конкретный алгоритм вычисления следующего прыжка маршрутизации гиперкубов, описанный в 2.1.5, потенциально может быть заменен другим алгоритмом при условии, что он удовлетворяет определенным свойствам. Одним из этих свойств является *оптимальность Хэмминга, означающая, что расстояние Хэмминга (L_1) от ξ* до η равно сумме расстояний Хэмминга от ξ до NextHop(ξ,η) и от NextHop(ξ,η) до η:

$$\|\xi - \eta\|_1 = \|\xi - \text{NextХоп}(\xi, \eta)\|_1 + \|\text{СледующийХоп}(\xi, \eta) - \eta\|_1 \qquad (14)$$

Здесь kξ – ηk_1 *— расстояние Хэмминга* между ξ и η, *равное числу битовых позиций, в которых* ξ и η отличаются:[15]

$$||\xi - \eta||_1 = \sum_i |\xi_i - \eta_i| \qquad (15)$$

Обратите внимание, что в целом следует ожидать только неравенства в (14), вытекающего из неравенства треугольника для L_1*-метрики. Оптимальность Хэмминга по существу означает, что NextHop(ξ,η)* лежит на одном из кратчайших путей (Хэмминга) от ξ до η. *Это также можно выразить, сказав, что* ν = NextHop(ξ,η) всегда получается из ξ путем изменения значений битов в некоторых позициях на их значения в η: *для любой битовой позиции* i *мы имеем* $\nu_i = \xi_i$ или $\nu_i = \eta_i$.[16]

2.1.9. Остановка NextHop. Другим важным свойством NextHop является его *остановка, означающая, что NextHop(ξ,η) =* ξ возможен только тогда, когда

[15] Когда задействованные адреса имеют разную длину (например, потому что они принадлежат разным рабочим цепочкам), следует учитывать только первые 96 бит адресов в приведенной выше формуле.

[16] Вместо оптимальности Хэмминга мы могли бы рассмотреть эквивалентное свойство *оптимальности Кадемлии, записанное для расстояния Кадемлия (или взвешенного L1), заданного* $||\xi - \eta||_K := \sum_i 2^{-i}|\xi_i - \eta_i|$ вместо расстояния Хэмминга.

$\xi = \eta$. Другими словами, если мы еще не достигли η, следующий прыжок не может совпадать с нашей нынешней позицией.

Это свойство подразумевает, что путь от ξ до η, *т. е. последовательность промежуточных адресов* $\xi^{(0)} := \xi$, $\xi^{(n)} := \text{NextHop}(\xi^{(n-1)},\eta)$, будет постепенно стабилизироваться при η: *для некоторого* $N \geq 0$ мы имеем $\xi^{(n)} = \eta$ для всех $n \geq N$. *Действительно, всегда можно взять* $N := ||\xi - \eta||_1$.

2.1.10. Выпуклость пути HR по отношению к шардингу. Следствием свойства оптимальности Хэмминга (14) является то, что мы называем *выпуклостью* пути от ξ к η относительно шардинга. А именно, если $\xi^{(0)} := \xi$, $\xi^{(n)} := \text{NextHop}(\xi^{(n-1)},\eta)$ — вычисляемый путь от ξ до η, *а* N — первый индекс, такой, что $\xi^{(N)} = \eta$, *а* S — осколок некоторой рабочей цепи в любой конфигурации сегмента, то индексы i с $\xi^{(i)}$, находящиеся в сегменте S, составляют подинтервал в $[0,H]$. Другими словами, если целые числа $0 \leq i \leq j \leq k \leq N$ таковы, что $\xi^{(i)}, \xi^{(k)} \in S$, *то* $\xi^{(j)}$ также $\in S$.

Это свойство выпуклости важно для некоторых доказательств, связанных с пересылкой сообщений при наличии динамического сегментирования.

2.1.11. Внутренняя маршрутизация. Обратите внимание, что адрес следующего прыжка, вычисляемый в соответствии с правилами, определенными в разделе 2.1.5, может принадлежать к той же цепочке шардчейна, что и текущий (т.е. тот, который содержит транзитный адрес). В этом случае «внутренняя маршрутизация» происходит немедленно, транзитный адрес заменяется значением вычисляемого адреса следующего прыжка, а шаг вычисления адреса следующего прыжка повторяется до тех пор, пока не будет получен адрес следующего прыжка, лежащий за пределами текущей шардчейна. Затем сообщение хранится в выходной очереди транзита в соответствии с его вычисляемым адресом следующего прыжка, причем его последний вычисляемый транзитный адрес является «промежуточным владельцем» транзитного сообщения. Если текущий shardchain разделяется на две shardchains перед дальнейшей пересылкой сообщения, то именно shardchain, содержащий промежуточный владелец, наследует это транзитное сообщение.

Кроме того, мы можем продолжить вычисление адресов следующего прыжка только для того, чтобы узнать, что адрес назначения уже принадлежит текущему шардчейну. В этом случае сообщение будет обработано (транзакцией) внутри этого шардчейна, а не перенаправлено дальше.

2.1.12. Соседние шардчейны. Два сегмента в конфигурации сегмента — или два соответствующих шардчейна — считаются *соседями или соседними шардчейнами, если один из них содержит адрес следующего прыжка по крайней мере для одной комбинации разрешенных адресов источника и назначения, а другой содержит транзитный адрес для той же комбинации. Другими словами, две шардчейны являются соседями, если сообщение может быть перенаправлено непосредственно из одного из них в другой через Hypercube Routing.*

Мастерчейн также включен в это определение, как если бы это был единственный шардчейн рабочей цепи с *workchain_id* = −1. В этом отношении он является соседом всех остальных шардчейнов.

2.1.13. Любой осколок является соседом самого себя. Обратите внимание, что шардчейн всегда считается соседом самого себя. Это может показаться излишним, потому что мы всегда повторяем вычисления следующего прыжка, описанные в 2.1.5, пока не получим адрес следующего прыжка за пределами текущего шардчейна (см. 2.1.11). Однако есть, по крайней мере, две причины для такой договоренности:

- Некоторые сообщения имеют адрес источника и назначения внутри одной и той же цепочки сегментов, по крайней мере, при создании сообщения. Однако, если такое сообщение не обрабатывается сразу в том же блоке, где оно было создано, оно должно быть добавлено в очередь исходящих сообщений его shardchain и импортировано как входящее сообщение (с записью в *InMsgDescr) в один из последующих блоков того же shardchain.*[17]

[17] Обратите внимание, что вычисления next-hop и внутренней маршрутизации по-прежнему применяются к таким сообщениям, так как текущая shardchain может быть

- Кроме того, адрес следующего прыжка может первоначально находиться в какой-то другой шардчейне, которая позже объединяется с текущей шардчейн, так что следующий прыжок становится внутри той же цепочки. Затем сообщение должно быть импортировано из очереди исходящих сообщений объединенной shardchain и перенаправлено или обработано в соответствии с его адресом nexthop, даже если они теперь находятся внутри одной и той же цепочки сегментов.

2.1.14. Гиперкубовая маршрутизация и интернет-провайдер. В конечном счете, здесь применима парадигма бесконечного шардинга (ISP): шардчейн следует рассматривать как временное объединение цепей счетов, сгруппированных вместе исключительно для минимизации генерации блоков и накладных расходов на передачу.

Пересылка сообщения проходит через несколько промежуточных цепей учетных записей, некоторые из которых могут лежать в одном и том же сегменте. В этом случае, как только сообщение достигает цепочки учетных записей, лежащей в этом сегменте, оно немедленно («внутренне») направляется внутрь этого сегмента до тех пор, пока не будет достигнута последняя цепочка учетных записей, лежащая в том же сегменте (см. 2.1.11). Затем сообщение помещается в очередь вывода этой последней цепочки учетных записей.[18]

2.1.15. Представление транзитных и последующих адресов. Обратите внимание, что адреса транзита и следующего прыжка отличаются от адреса источника только в *workchain_id* и в первых (наиболее значимых) 64 битах адреса учетной записи. Поэтому они могут быть представлены 96-битными строками. Кроме того их *workchain_id* обычно совпадает с *workchain_id* либо адреса источника, либо адреса назначения; для обозначения этой ситуации

разделена перед обработкой сообщения. В этом случае новая подшардцепь, содержащая адрес назначения, унаследует сообщение.

[18] Мы можем определить (виртуальную) очередь вывода учетной записи (цепочки) как подмножество *OutMsgQueue* сегмента, в настоящее время содержащего эту учетную запись, которая состоит из сообщений с транзитными адресами, равными адресу учетной записи.

может использоваться пара бит, что еще больше сокращает пространство, необходимое для представления транзитных и следующих адресов.

Фактически, требуемое хранилище может быть сокращено еще больше, если заметить, что конкретный алгоритм маршрутизации гиперкубов, описанный в 2.1.5, всегда генерирует промежуточные (т.е. транзитные и следующие) адреса, которые совпадают с адресом назначения в их первых k битах и с адресом источника в их оставшихся битах. Поэтому можно использовать только значения $0 \le ktr, knh \le 96$, чтобы полностью указать адреса транзита и следующего прыжка. Можно также заметить, что $k' := k_{nh}$ оказывается фиксированной функцией $k := k_{tr}$ (например, $k' = k + \mathrm{n}_2 = k + 4$ для $k \ge 32$), и поэтому включать в сериализацию только одно 7-битное значение k.

Такие оптимизации имеют очевидный недостаток, заключающийся в том, что они слишком полагаются на конкретный используемый алгоритм маршрутизации, который может быть изменен в будущем, поэтому они используются в 3.1.15 с положением о том, чтобы при необходимости указать более общие промежуточные адреса.

2.1.16. Конверты с сообщениями. Адреса транзита и следующего прыжка пересылаемого сообщения не включаются в само сообщение, а хранятся в специальном конверте *сообщения, который представляет собой ячейку (или фрагмент ячейки), содержащую адреса транзита и следующего прыжка с вышеуказанными оптимизациями, некоторую другую информацию, относящуюся к пересылке и обработке, и ссылку на ячейку, содержащую неизмененное исходное сообщение. Таким образом, сообщение может быть легко «извлечено» из исходного конверта (например, того, который присутствует в InMsgDescr) и помещено в другой конверт (например, перед включением в OutMsgQueue).*

При представлении блока в виде дерева, или, скорее, группы обеспечения доступности баз данных, двух различных конвертов будут содержать ссылки на общую ячейку с исходным сообщением. Если сообщение большое, такое расположение позволяет избежать необходимости хранить более одной копии сообщения в блоке.

2.2 Протокол маршрутизации Hypercube

В этом разделе раскрываются подробности протокола маршрутизации гиперкубов, используемого блокчейном TON для обеспечения гарантированной доставки сообщений между смарт-контрактами, находящимися в произвольных шардчейнах. Для целей этого документа мы будем ссылаться на вариант маршрутизации гиперкубов, используемый блокчейном TON, как Hypercube Routing (HR).

2.2.1. Уникальность сообщения. Прежде чем продолжить, давайте заметим, что любое (внутреннее) сообщение *уникально. Напомним, что сообщение содержит полный адрес источника вместе со временем логического создания, а все исходящие сообщения, созданные одним и тем же смарт-контрактом, имеют строго увеличенное время логического создания (см. 1.4.6); поэтому сочетание полного адреса источника и логического времени создания однозначно определяет сообщение. Поскольку мы предполагаем, что выбранная хэш-функция sha256 устойчива к коллизиям, сообщение однозначно определяется его хэшем, поэтому мы можем идентифицировать два сообщения, если знаем, что их хэши совпадают.*

Это не распространяется на внешние сообщения «из ниоткуда», которые не имеют исходных адресов. Особое внимание следует уделять предотвращению повторных атак, связанных с такими сообщениями, особенно со стороны разработчиков смарт-контрактов кошельков пользователей. Одним из возможных решений является включение порядкового номера в тело таких сообщений и сохранение количества внешних сообщений, уже обработанных внутри постоянных данных смарт-контракта, отказываясь обрабатывать внешнее сообщение, если его порядковый номер отличается от этого количества.

2.2.2. Идентификация сообщений с равными хэшами. Блокчейн TON предполагает, что два сообщения с одинаковыми хэшами совпадают, и рассматривает любое из них как избыточную копию другого. Как объяснялось выше в разделе 2.2.1, это не приводит к каким-либо неожиданным последствиям для внутренних сообщений. Однако, если отправить два совпадающих «сообщения из ниоткуда» на смарт-контракт, может случиться так, что будет доставлено только одно из них — или оба. Если их действие не должно быть идемпотентным (т.е. если обработка сообщения дважды имеет эффект, отличный от его обработки один раз), следует предусмотреть некоторые положения для различения этих двух сообщений, например, путем включения в них порядкового номера.

В частности, *InMsgDescr* и *OutMsgDescr* используют хэш (непробеленного) сообщения в качестве ключа, молчаливо предполагая, что отдельные сообщения имеют разные хэши. Таким образом, можно проследить путь и судьбу сообщения через разные цепочки шардов, посмотрев хэш сообщения в *InMsgDescr* и *OutMsgDescr* разных блоков.

2.2.3. Структура *OutMsgQueue. Напомним, что исходящие сообщения — как созданные внутри shardchain, так и транзитные сообщения, ранее* импортированные из соседнего shardchain для ретрансляции в шардчейн nexthop, — накапливаются в *OutMsgQueue, который является частью состояния* shardchain (см. 1.2.7). В отличие от *InMsgDescr* и *OutMsgDescr, ключом в OutMsgQueue* является не хэш сообщения, а его адрес следующего прыжка — или, по крайней мере, его первые 96 бит — сцепленные с хэшем сообщения.

Кроме того, *OutMsgQueue* - это не просто словарь (hashmap), отображающий свои ключи в (конвертированные) сообщения. Скорее, это минимальный расширенный *словарь в отношении логического времени создания, означающий, что каждый узел дерева Patricia, представляющий OutMsgQueue,* имеет дополнительное значение (в данном случае беззнаковое 64-битное целое число), и что это значение дополнения в каждом узле форка установлено равным минимальному значению дополнений его дочерних элементов. Значение дополнения листа равно логическому времени создания сообщения, содержащегося в этом листе; его не обязательно хранить явно.

2.2.4. Осмотр *OutMsgQueue* соседа. Такая структура для *OutMsgQueue* позволяет валидаторам соседнего шардчейна проверять его, чтобы найти его часть (поддерево Patricia), относящуюся к ним (т. е. состоящую из сообщений с адресом следующего прыжка, принадлежащим соседнему сегменту, о котором идет речь, или имеющему адрес следующего прыжка с заданным двоичным префиксом), а также быстро вычислить «самый старый» (т.е. с минимальным логическим временем создания) сообщение в этой части.

Кроме того, валидаторам сегментов даже не нужно отслеживать общее состояние всех соседних цепей шардов — им нужно только сохранить и обновить копию своего *OutMsgQueue или даже его поддерева, связанного с ними.*

2.2.5. Логическое время монотонности: импорт самого старого сообщения от соседей. Первое фундаментальное локальное условие пересылки сообщений, называемое *(импорт сообщения) (логическое время) условием монотонности, можно резюмировать следующим образом:*

> При импорте сообщений в *InMsgDescr* блока shardchain из *OutMsgQueues соседних shardchains валидаторы должны импортировать сообщения в порядке возрастания их логического времени; в случае привязки сообщение с меньшим хэшем импортируется первым.*

Точнее, каждый блок shardchain содержит хэш блока masterchain (считается «последним» блоком masterchain на момент создания блока shardchain), который, в свою очередь, содержит хэши самых последних блоков shardchain. Таким образом, каждый блок шардчейнов косвенно «знает» самое последнее состояние всех других шардчейнов, и особенно соседних с ним шардчейнов, включая их *OutMsgQueues.*[19]

Теперь альтернативная эквивалентная формулировка условия монотонности выглядит следующим образом:

[19] В частности, если хэш недавнего блока соседнего шардчейна еще не отражен в последнем блоке мастерчейна, то его модификации *OutMsgQueue* не должны приниматься во внимание.

> Если сообщение импортируется в *InMsgDescr* нового блока, время его логического создания не может быть больше, чем время любого сообщения, оставшегося неимпортированным в *OutMsgQueue* самого последнего состояния любого из соседних сегментов.

Именно эта форма условия монотонности появляется в локальных условиях согласованности блоков TON Blockchain и обеспечивается валидаторами.

2.2.6. Свидетели нарушений сообщения импортируют условие монотонности логического времени. Обратите внимание, что если это условие не выполняется, может быть построено небольшое доказательство Меркла, свидетельствующее о его сбое. Такое доказательство будет содержать:

- Путь в *OutMsgQueue* соседа от корня к определенному сообщению m с небольшим временем логического создания.
- Путь в *InMsgDescr* рассматриваемого блока, показывающий, что ключ, равный Hash(m), отсутствует в *InMsgDescr* (т.е. что m не был включен в текущий блок).
- Доказательство того, что m не было включено в предыдущий блок того же шардчейна, используя информацию заголовка блока, содержащую наименьшее и наибольшее логическое время всех сообщений, импортированных в блок (см. 2.3.4–2.3.7 для получения дополнительной информации).
- Путь в *InMsgDescr* к другому включенному сообщению $m0$, *такой, что либо* $Lt(m') >$ Lt(m), либо Lt(m') = Lt(m) и Hash(m') > Hash(m).

2.2.7. Удаление сообщения из *OutMsgQueue. Рано или поздно сообщение должно быть удалено из OutMsgQueue*; в противном случае хранилище, используемое *OutMsgQueue*, вырастет до бесконечности. С этой целью вводится несколько «правил сбора мусора». Они позволяют удалять сообщение из *OutMsgQueue* во время оценки блока только в том случае, если в *OutMsgDescr* этого блока присутствует явная специальная «запись доставки». Эта запись содержит либо ссылку на соседний блок shardchain,

который включил сообщение в свой *InMsgDescr* (хэш блока достаточен, но собранный материал для блока может содержать соответствующее доказательство Меркла), либо доказательство Меркла о том, что сообщение было доставлено в конечный пункт назначения через instant Hypercube Routing.

2.2.8. Гарантированная доставка сообщений по Hypercube Routing. Таким образом, сообщение не может быть удалено из очереди исходящих сообщений, если оно не было либо ретранслировано в шардчейн следующего прыжка, либо доставлено в конечный пункт назначения (см. 2.2.7). Между тем, условие монотонности импорта сообщений (см. 2.2.5) гарантирует, что любое сообщение рано или поздно будет ретранслировано в следующую цепочку шардчейнов, принимая во внимание другие условия, которые требуют, чтобы валидаторы использовали не менее половины пространства блока или ограничения газа для импорта входящих внутренних сообщений (в противном случае валидаторы могут создать пустые блоки или импортировать только внешние сообщения даже при наличии непустых очередей исходящих сообщений у своих соседей).

2.2.9. Порядок обработки сообщений. Когда несколько импортированных сообщений обрабатываются транзакциями внутри блока, *условия порядка обработки сообщений* гарантируют, что старые сообщения обрабатываются первыми. Точнее, если блок содержит две транзакции t и t' одного и того же аккаунта, которые обрабатывают входящие сообщения m и $m0$ *соответственно и* $Lt(m) <$ Lt(m'), то у нас должны быть Lt(t) < Lt(t').

2.2.10. FIFO гарантирует маршрутизацию Hypercube. Условия заказа на обработку сообщений (см. 2.2.9), наряду с условиями монотонности импорта сообщений (см. 2.2.5), подразумевают *гарантии FIFO для маршрутизации Hypercube. А именно, если смарт-контракт* ξ создает два сообщения m и m' с одинаковым *назначением* η, *а* m' генерируется позже m (имеется в виду, что $m \prec m'$, *следовательно,* $Lt(m) <$ Lt(m')), то m будет обработан η перед m'. *Это связано с тем, что оба сообщения будут следовать* одним и тем же шагам маршрутизации по пути от ξ к η (алгоритм маршрутизации Hypercube, описанный в 2.1.5, является

детерминированным), и во всех исходящих очередях и описаниях входящих сообщений m' будет отображаться «после» m.[20]

Если сообщение m' может быть доставлено в B через мгновенную маршрутизацию Hypercube, это уже не обязательно так. Таким образом, простой способ обеспечения дисциплины доставки сообщений FIFO между парой смарт-контрактов заключается в установке специального бита в заголовке сообщения, предотвращающего его доставку через ММСП.

2.2.11. Гарантия уникальности доставки Hypercube Routing. Обратите внимание, что условия монотонности импорта сообщений также подразумевают *уникальность* доставки любого сообщения через Hypercube Routing, то есть то, что оно не может быть импортировано и обработано смарт-контрактом назначения более одного раза. Позже в 2.3 мы увидим, что обеспечение уникальности доставки, когда активны как Hypercube Routing, так и Instant Hypercube Routing, сложнее.

2.2.12. Обзор маршрутизации Hypercube. Давайте суммируем все шаги маршрутизации, выполненные для доставки внутреннего сообщения m, созданного исходной учетной записью ξ_0, в целевую учетную запись η. *Обозначаем* ξ_{k+1} := NextHop(ξ_k,η), k = 0,1,2,… промежуточные адреса, продиктованные HR для пересылки сообщения m в конечный пункт назначения η. *Пусть* S_k — осколок, содержащий ξ_k.

- [Рождение] — Сообщение m с *целевым* η создается транзакцией t, принадлежащей учетной записи ξ_0, находящейся в какой-то shardchain S_0. *Время логического создания* $Lt(m)$ фиксируется в этой точке и включается в сообщение m.

- [ImmediateProcessing?] — Если *целевой* η находится в той же цепочке S_0, *сообщение может быть обработано в том же блоке, в котором оно было сгенерировано. В этом случае* m включается в *OutMsgDescr* с флагом, указывающим, что он был обработан в этом самом блоке и не нуждается в дальнейшей пересылке. Другая копия m включена в *InMsgDescr вместе с обычными данными, описывающими*

[20] Это утверждение не так тривиально, как кажется на первый взгляд, потому что некоторые из шардчейнов

обработку входящих сообщений. (Обратите внимание, что m не входит в *OutMsgQueue* S_0*.)*

- [InitialInternalRouting] — Если m либо имеет назначение вне S_0*, либо не обрабатывается в том же блоке, где он был сгенерирован, применяется внутренняя процедура маршрутизации, описанная в 2.1.11, до тех пор, пока не будет найден индекс k,* такой, что ξ_k лежит в S_0*, но* ξ_{k+1} = NextHop(ξk,η) этого не делает (т.е. $S_k = S_0$*, но* $S_{k+1} \neq S_0$*). Кроме того, этот процесс останавливается, если* $\xi_k = \eta$ или ξ_k совпадает с η в его первых 96 битах.
- [OutboundQueuing] — Сообщение m включено в *OutMsgDescr* (с ключом, равным его хэшу), с конвертом, содержащим его транзитный адрес ξ_k и адрес следующего прыжка ξ_{k+1}, как описано в 2.1.16 и 2.1.15. Это же сообщение в оболочке также включено в *OutMsgQueue* состояния S_k*, с ключом, равным сцеплению первых 96 бит его адреса следующего прыжка* ξ_{k+1} (который может быть равен η если η принадлежит S_k*) и хэша сообщения Hash(m).*
- [QueueWait] — Сообщение m ожидает дальнейшей пересылки в *OutMsgQueue* shardchain S_k. В то же время shardchain S_k может

вовлеченные могут разделяться или объединяться во время маршрутизации. Правильное доказательство может быть получено путем принятия точки зрения ISP для HR, как описано в 2.1.14, и наблюдения за тем, что *m0* всегда будет отставать от *m, либо с точки зрения достигнутой промежуточной цепочки счетов, либо, если они окажутся в одной и той же цепочке счетов, с точки зрения логического времени создания.*

Важным наблюдением является то, что «в любой заданный момент времени» (логически; более точным описанием было бы «в общем состоянии, полученном после обработки любого причинно замкнутого подмножества *F* блоков»), промежуточные цепочки счетов, принадлежащие одному и тому же осколку, являются смежными на пути от ξ до η (т.е. не может иметь связей счетов, принадлежащих какому-то другому осколку между ними). Это «свойство выпуклости» (ср. 2.1.10) алгоритма маршрутизации Hypercube, описанного в 2.1.5.

разделяться или сливаться с другими shardchains; в этом случае новый сегмент S'_k, содержащий транзитный адрес ξ_k, наследует m в своем *OutMsgQueue.*

- [ImportInbound] — В какой-то момент в будущем валидаторы для shardchain S_{k+1}, содержащего адрес следующего прыжка ξ_{k+1}, сканируют *OutMsgQueue* в состоянии shardchain S_k и решают импортировать сообщение m в соответствии с условием монотонности (см. 2.2.5) и другими условиями. Генерируется новый блок для shardchain S_{k+1}, с конвертированной копией m, включенной в его *InMsgDescr. Запись в InMsgDescr* содержит также причину импорта m в этот блок, с хэшем самого последнего блока shardchain S'_k, и предыдущими адресами следующего прыжка и транзита ξ_k и ξ_{k+1}, *так что соответствующая запись в OutMsgQueue* из S'_k может быть легко найдена.

- [Подтверждение] — Эта запись в *InMsgDescr* S_{k+1} также служит подтверждением для S'_k. В более позднем блоке S'_k сообщение m должно быть удалено из *OutMsgQueue* S'_k; это изменение отражается в специальной записи в *OutMsgDescr* блока S'_k, который выполняет это изменение состояния.

  [Forwarding?] — Если конечный пункт назначения η m не находится в S_{k+1}, *сообщение пересылается. Маршрутизация гиперкубов применяется до тех пор, пока* получаются некоторые ξ_l, $l > k$ *и* ξ_{l+1} = NextHop(ξ_l,η), так что ξ_l лежит в S_{k+1}, *а* ξ_{l+1} — нет (см. 2.1.11). После этого недавно разработанная копия m с транзитным адресом, установленным на ξ_l, и адресом следующего прыжка ξ_{l+1} включается как в *OutMsgDescr* текущего блока S_{k+1}, так и в *OutMsgQueue* нового состояния S_{k+1}. *Запись* m в *InMsgDescr* содержит флаг, указывающий, что сообщение было переслано; запись в *OutMsgDescr* содержит сообщение с новым конвертом и флаг, указывающий, что это пересылаемое сообщение. Затем все шаги, начинающиеся с [OutboundQueueing], повторяются, для l вместо k.

- [Processing?] — Если конечный пункт назначения η m находится в S_{k+1}, *то блок* S_{k+1}, импортировавший сообщение, должен обработать его транзакцией t, включенной в тот же блок. В этом случае *InMsgDescr* содержит ссылку на t по своему логическому времени Lt(t) и флаг, указывающий, что сообщение было обработано.

Приведенный выше алгоритм маршрутизации сообщений не учитывает некоторые дополнительные изменения, необходимые для реализации мгновенной гиперкубической маршрутизации (IHR). Например, сообщение может быть *отброшено* после импорта (перечисленного в *InMsgDescr) в его конечный или промежуточный блок шардчейна, поскольку представлено доказательство доставки через ММСП в конечный пункт назначения. В этом случае такое доказательство должно быть включено в InMsgDescr,* чтобы объяснить, почему сообщение не было отправлено дальше или обработано.

2.3 Мгновенная маршрутизация Hypercube и комбинированные гарантии доставки

В этом разделе описывается протокол мгновенной гиперкубовой маршрутизации, обычно применяемый TON Blockchain параллельно с ранее обсуждавшимся протоколом маршрутизации Hypercube для достижения более быстрой доставки сообщений. Однако, когда и Hypercube Routing, и Instant Hypercube Routing применяются к одному и тому же сообщению параллельно, достижение доставки и уникальных гарантий доставки сложнее. Эта тема также обсуждается в этом разделе.

2.3.1. Обзор мгновенной маршрутизации гиперкубов. Поясним основные шаги, применяемые при применении к сообщению механизма мгновенной гиперкубической маршрутизации (ММСП). (Обратите внимание, что обычно и обычные HR, и ММСП работают параллельно для одного и того же сообщения; некоторые положения должны быть приняты, чтобы гарантировать уникальность доставки любого сообщения.)

Рассмотрим маршрутизацию и доставку одного и того же сообщения m с источником ξ и η назначения , как описано в 2.2.12:

- [NetworkSend] — После того, как валидаторы S_0 согласовали и подписали блок, содержащий создающую транзакцию t для m, *и заметили, что* η *назначения* m не находится внутри S_0, *они могут отправить дейтаграмму (зашифрованное сетевое сообщение), содержащую сообщение* m вместе с доказательством Merkle о его включении в *OutMsgDescr* только что сгенерированного блока в группу валидаторов shardchain T, в настоящее время владеющего целевым η.

- [NetworkReceive] — Если валидаторы shardchain T получают такое сообщение, они проверяют его действительность, начиная с самого последнего блока masterchain и перечисленных в нем хэшей блока shardchain, включая самый последний «канонический» блок shardchain S_0. Если сообщение недопустимо, они молча отбрасывают его. Если этот блок shardchain S_0 имеет больший порядковый номер, чем тот, который указан в последнем блоке masterchain, они могут либо отбросить его, либо отложить проверку до появления следующего блока masterchain.

- [InclusionConditions] — Валидаторы проверяют условия включения сообщения m. *В частности, они должны проверить, что это сообщение не было доставлено ранее, и что OutMsgQueues соседей не имеют необработанных исходящих сообщений с назначениями в* T с меньшим временем логического создания, чем Lt(m).

- [Deliver] — Валидаторы доставляют и обрабатывают сообщение, включая его в *InMsgDescr* текущего блока shardchain вместе с битом, указывающим, что это сообщение IHR, доказательство Меркла о его включении в *OutMsgDescr* исходного блока и логическое время транзакции t', обрабатывающей это входящее сообщение в текущий сгенерированный блок.

- [Confirm] — Наконец, валидаторы отправляют зашифрованные дейтаграммы всем группам валидаторов промежуточных шардчейнов на пути от ξ до η, *содержащие доказательство Меркла о включении сообщения* m в *InMsgDescr* его конечного назначения.

Валидаторы промежуточной цепочки сегментов могут использовать это доказательство для *удаления* копии сообщения m путешествует по правилам HR, импортируя сообщение в свой *InMsgDescr* вместе с доказательством Окончательной доставки Merkle и устанавливая флаг, указывающий, что сообщение было отброшено.

Общая процедура еще проще, чем для Hypercube Routing. Обратите внимание, однако, что ММСП поставляются без доставки или гарантий FIFO: сетевая датаграмма может быть потеряна при передаче, или валидаторы целевого шардчейна могут решить не действовать по ней, или они могут отклонить ее из-за переполнения буфера. Именно по этой причине ММСП используются в качестве дополнения к ЛР, а не в качестве замены.

2.3.2. Общие гарантии возможной поставки. Обратите внимание, что сочетание HR и IHR гарантирует конечную доставку любого внутреннего сообщения в конечный пункт назначения. Действительно, HR сам по себе гарантированно доставит любое сообщение в конечном итоге, и HR для сообщения m может быть отменен на промежуточном этапе только доказательством Merkle о доставке m в конечный пункт назначения (через IHR).

2.3.3. Общие уникальные гарантии доставки. Однако *уникальность* доставки сообщений для сочетания HR и ММСП труднее достичь. В частности, необходимо проверить следующие условия, и, при необходимости, иметь возможность предоставить краткие доказательства Меркла, что они содержатся или не содержатся:

- Когда сообщение m импортируется в следующий промежуточный блок shardchain через HR, мы должны проверить, что m еще не был импортирован через HR.
- Когда m импортируется и обрабатывается в конечном пункте назначения shardchain, мы должны проверить, что m еще не был обработан. Если да, то есть три подкафайла:

– Если m рассматривается для импорта через HR, и он уже был импортирован через HR, он не должен быть импортирован вообще.

– Если m рассматривается для импорта через HR, и он уже был импортирован через ММСП (но не HR), то он должен быть импортирован и немедленно выброшен (без обработки транзакцией). Это необходимо для удаления m из *OutMsgQueue* его предыдущего промежуточного шардчейна.

– Если m рассматривается для импорта через ММСП и он уже был импортирован либо через ММСП, либо через HR, он не должен быть импортирован вообще.

2.3.4. Проверка того, было ли сообщение уже доставлено в конечный пункт назначения. Рассмотрим следующий общий алгоритм проверки того, было ли сообщение m уже доставлено в конечный пункт назначения η: *Можно просто отсканировать последние несколько блоков, принадлежащих shardchain, содержащих адрес назначения, начиная с последнего блока и работая в обратном направлении через предыдущие ссылки на блок. (Если есть два предыдущих блока, т. е. если в какой-то момент произошло событие слияния shardchain, вы будете следовать по цепочке, содержащей адрес назначения.) InMsgDescr* каждого из этих блоков можно проверить на наличие записи с ключом Hash(m). Если такая запись найдена, сообщение m уже доставлено, и мы можем легко построить доказательство Меркла этого факта. Если мы не найдем такую запись до прибытия в блок B с Lt+(B) < Lt(m), подразумевая, что m не может быть доставлен в B или любом из его предшественников, то сообщение m определенно еще не доставлено.

Очевидным недостатком этого алгоритма является то, что, если сообщение m очень старое (и, скорее всего, доставленное давным-давно), что означает, что оно имеет небольшое значение Lt(m), то перед выдачей ответа необходимо будет отсканировать большое количество блоков. Кроме того, если ответ отрицательный, размер доказательства Меркла этого факта будет линейно увеличиваться с количеством сканируемых блоков.

2.3.5. Проверка того, было ли сообщение ММСП уже доставлено в конечный пункт назначения. Чтобы проверить, было ли сообщение ММСП m уже доставлено в шардчейн назначения, мы можем применить общий алгоритм, описанный выше (см. 2.3.4), модифицированный для проверки только последних блоков c на наличие некоторой небольшой константы c (скажем, $c = 8$). Если после проверки этих блоков не удается прийти к какому-либо выводу, то валидаторы для целевого шардчейна могут просто отбросить сообщение ММСП вместо того, чтобы тратить больше ресурсов на эту проверку.

2.3.6. Проверка того, было ли hr-сообщение уже доставлено через HR в конечный пункт назначения или промежуточный шардчейн. Чтобы проверить, было ли уже импортировано через HR полученное сообщение m (или, скорее, сообщение m, рассматриваемое для импорта через HR), мы можем использовать следующий алгоритм: Пусть ξ_k — транзитный адрес m (принадлежащий соседнему shardchain Sk), а ξ_{k+1} — его адрес следующего прыжка (принадлежащий рассматриваемому шардчейну). Поскольку мы рассматриваем возможность включения m, m должно присутствовать в *OutMsgQueue* самого последнего состояния shardchain S_k, *причем* ξ_k и ξ_{k+1} указаны в его конверте. В частности, (a) сообщение было включено в *OutMsgQueue, и мы можем даже знать, когда, потому что запись в OutMsgQueue* иногда содержит логическое время блока, в который оно было добавлено, и (b) оно еще не было удалено из *OutMsgQueue*.

Теперь валидаторы соседнего шардчейна должны удалить сообщение из *OutMsgQueue,* как только они заметят, что сообщение (с адресами транзита и следующего прыжка ξ_k и ξ_{k+1} в конверте) было импортировано в *InMsgDescr* шардчейна следующего прыжка сообщения. Таким образом, (b) подразумевает, что сообщение могло быть импортировано в *InMsgDescr* предыдущего блока только в том случае, если этот предыдущий блок является очень новым (т. е. еще не известен самому последнему соседнему блоку shardchain). Поэтому только очень ограниченное количество предшествующих блоков (обычно один или два, самое большее) должно быть проверено алгоритмом, описанным в 2.3.4, чтобы сделать вывод о

том, что сообщение еще не было импортировано.[21] Фактически, если эта проверка выполняется самими валидаторами или коллаторами для текущего шардчейна, ее можно оптимизировать, сохранив в памяти *InMsgDescrs нескольких последних блоков.*

2.3.7. Проверка того, было ли сообщение HR уже доставлено через ММСП в конечный пункт назначения. Наконец, чтобы проверить, было ли сообщение HR уже доставлено в конечный пункт назначения с помощью ММСП, можно использовать общий алгоритм, описанный в 2.3.4. В отличие от 2.3.5, мы не можем прервать процесс проверки после сканирования фиксированного количества последних блоков в целевой shardchain, потому что HR-сообщения не могут быть удалены без причины.

Вместо этого мы косвенно ограничили количество проверяемых блоков, запретив включение сообщения ММСП m в блок B его целевой шардчейна, если в шардчейне назначения уже больше, чем, скажем, $c = 8$ блоков B' с Lt+(B') ≥ Lt(m).

Такое условие фактически ограничивает временной интервал после создания сообщения m, в течение которого оно могло бы быть доставлено с помощью ММСП, так что потребуется проверить лишь небольшое количество блоков шардчейна назначения (не более *c).*

Обратите внимание, что это условие хорошо согласуется с модифицированным алгоритмом, описанным в 2.3.5, фактически запрещая валидаторам импортировать вновь полученное сообщение ММСП, если *более c* = 8 шагов.для проверки того, что оно еще не было импортировано, требуется

[21] Необходимо не только искать ключ Hash(m) в *InMsgDescr* этих блоков, но и проверять промежуточные адреса в конверте соответствующей записи, если они найдены.

3 Сообщения, дескрипторы сообщений и очереди

В этой главе представлен внутренний макет отдельных сообщений, дескрипторов сообщений (таких как *InMsgDescr* или *OutMsgDescr) и очередей сообщений (например, OutMsgQueue). Здесь также обсуждаются сообщения в конвертах (см. 2.1.16).*

Обратите внимание, что большинство общих соглашений, связанных с сообщениями, должны соблюдаться всеми шардчейнами, даже если они не принадлежат к основной цепочке; в противном случае обмен сообщениями и взаимодействие между различными рабочими цепочками были бы невозможны. Именно *интерпретация* содержимого сообщения и *обработка* сообщений, обычно некоторыми транзакциями, различаются между рабочими цепочками.

3.1 Адрес, валюта и макет сообщения

Эта глава начинается с некоторых общих определений, за которыми следует точное расположение адресов, используемых для сериализации адресов источника и назначения в сообщении.

3.1.1. Некоторые стандартные определения. Для удобства читателя мы воспроизводим здесь несколько общих определений TL-B. [22] Эти определения используются ниже при обсуждении адреса и макета сообщения, но в остальном не связаны с блокчейном TON.

```
unit$_ = Unit;
true$_ = True;
// EMPTY False;
bool_false$0 = Bool;
bool_true$1 = Bool;
nothing$0 {X:Type} = Maybe X;
just$1 {X:Type} value:X = Maybe X;
```

[22] Описание более старой версии TL можно найти на https://core.telegram.org/mtproto/TL. В качестве альтернативы неофициальное введение в схемы TL-B можно найти в [4, 3.3.4].

```
left$0 {X:Type} {Y:Type} value:X = Either X Y;
right$1 {X:Type} {Y:Type} value:Y = Either X Y;
pair$_ {X:Type} {Y:Type} first:X second:Y = Both X Y;

bit$_ _:(## 1) = Bit;
```

3.1.2. Схема TL-B для адресов. Сериализация адресов источника и назначения определяется следующей схемой TL-B:

```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 8) external_address:(len * Bit)
                        = MsgAddressExt;
anycast_info$_  depth:(## 5) rewrite_pfx:(depth * Bit) = Anycast;
addr_std$10 anycast:(Maybe Anycast)
    workchain_id:int8 address:uint256 = MsgAddressInt;
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9)
     workchain_id:int32 address:(addr_len * Bit) = MsgAddressInt;
_ MsgAddressInt = MsgAddress;
_ MsgAddressExt = MsgAddress;
```

Две последние строки определяют тип MsgAddress как внутреннее объединение типов MsgAddressInt и MsgAddressExt (не путать с их внешним объединением Either MsgAddressInt MsgAddressExt, как определено в 3.1.1), как если бы предыдущие четыре строки были повторены с правой стороной, замененной на MsgAddress. Таким образом, тип MsgAddress имеет четыре конструктора, а типы MsgAddressInt и MsgAddressExt являются подтипами MsgAddress.

3.1.3. Внешние адреса. Первые два конструктора, addr_none и addr_extern, используются для исходных адресов «сообщений из ниоткуда» (входящих внешних сообщений) и для адресов назначения «сообщений в никуда» (исходящих внешних сообщений). Конструктор addr_extern определяет «внешний адрес», который полностью игнорируется программным обеспечением TON Blockchain (которое рассматривает addr_extern как более длинный вариант addr_none), но может использоваться внешним программным обеспечением для своих целей. Например, специальный

внешний сервис может проверить адрес назначения всех исходящих внешних сообщений, найденных во всех блоках блокчейна TON, и, если в поле external_address присутствует специальное магическое число, проанализировать оставшуюся часть как IP-адрес и UDP-порт или адрес ADNL (TON Network) и отправить дейтаграмму с копией сообщения на полученный таким образом сетевой адрес.

3.1.4. Внутренние адреса. Два оставшихся конструктора, addr_std и addr_var, представляют внутренние адреса. Первый из них, addr_std, представляет собой подписанный 8-битный *workchain_id* (достаточный для мастерчейна и для базовой рабочей цепочки) и 256-битный внутренний адрес в выбранной рабочей цепочке. Второй из них, addr_var, представляет собой адреса в рабочих цепочках с «большой» *workchain_id, или внутренние адреса длиной, не равной 256. Оба этих конструктора* имеют необязательное значение anycast, отсутствующее по умолчанию, которое позволяет «перезаписать адреса» при наличии.[23]

По возможности средства проверки должны использовать addr_std вместо addr_var, но должны быть готовы принять addr_var во входящих сообщениях. Конструктор addr_var предназначен для будущих расширений.

Обратите внимание, что *workchain_id* должен быть допустимым идентификатором рабочей цепи, включенным в текущей конфигурации мастерчейна, а длина внутреннего адреса должна находиться в диапазоне, разрешенном для указанной рабочей цепочки. Например, нельзя использовать *workchain_id* = 0 (базовая рабочая цепочка) или *workchain_id* = -1 (мастерчейн) с адресами длиной не совсем 256 бит.

3.1.5. Представление сумм в валюте Грамм. Количества граммов выражаются с помощью двух типов, представляющих целые числа без знака переменной длины или знаки, плюс тип граммов, явно предназначенный

[23] *Переписывание адресов* — это функция, используемая для реализации «адресов anycast», используемых так называемыми *большими* или *глобальными* смарт-контрактами (см. [3, 2.3.18]), которые могут иметь экземпляры в нескольких шардчейнах. Когда переписывание адресов включено, сообщение может быть перенаправлено и обработано смарт-контрактом с адресом, совпадающим с адресом назначения до первых d *битов,* где $d \leq 32$ — «глубина расщепления» смарт-контракта, указанная в поле anycast.depth (см. 2.1.7). В противном случае адреса должны точно совпадать.

для представления неотрицательных количеств нанограммов, следующим образом:

```
var_uint$_ {n:#} len:(#< n) value:(uint (len * 8))
             = VarUInteger n;
var_int$_ {n:#} len:(#< n) value:(int (len * 8))
            = VarInteger n;
nanograms$_ amount:(VarUInteger 16) = Grams;
```

Если кто-то хочет представить x нанограмм, он выбирает целое число $e < 16$, так что $x < 2^{8e}$, и сериализует сначала e как беззнаковое 4-битное целое число, а затем *само* x как беззнаковое $8e$-битное целое число. Обратите внимание, что четыре нулевых бита представляют собой нулевое количество граммов.

Напомним (см. [3, A]), что первоначальный общий запас граммов зафиксирован на уровне пяти миллиардов (т. е. $5 \cdot 1018 < 263$ нанограмма) и, как ожидается, будет расти очень медленно. Поэтому все количества граммов, встречающиеся на практике, будут умещаться в беззнаковых или даже подписанных 64-битных целых числах. Валидаторы могут использовать 64-разрядное целочисленное представление граммов в своих внутренних вычислениях; Однако сериализация этих значений блокчейном — другое дело.

3.1.6. Представление коллекций произвольных валют. Напомним, что TON Blockchain позволяет своим пользователям определять произвольные криптовалюты или токены отдельно от Gram, при условии соблюдения некоторых условий. Такие дополнительные криптовалюты идентифицируются 32-битными *currency_ids. Список определенных дополнительных криптовалют является частью конфигурации блокчейна, хранящейся в мастерчейне.*

Когда необходимо представить несколько количеств одной или нескольких таких криптовалют, используется словарь (см. [4, 3.3]) с 32-битными *currency_ids в качестве ключей и значениями VarUInteger 32:*

```
extra_currencies$_ dict:(HashmapE 32 (VarUInteger 32))
                                = ExtraCurrencyCollection;
```

```
currencies$_ grams:Grams other:ExtraCurrencyCollection
                = CurrencyCollection;
```

Значение, прикрепленное к внутреннему сообщению, представлено значением типа CurrencyCollection, которое может описывать определенное (неотрицательное) количество (нано)граммов, а также некоторые дополнительные валюты, если это необходимо. Обратите внимание, что если дополнительные валюты не требуются, другие уменьшаются до одного нулевого бита.

3.1.7. Макет сообщения. Сообщение состоит из заголовка , за которым следует его *тело или полезная нагрузка. Тело по существу произвольно, чтобы интерпретироваться смарт-контрактом назначения. Заголовок сообщения является стандартным и организован следующим образом:*

```
int_msg_info$0 ihr_disabled:Bool bounce:Bool
   src:MsgAddressInt dest:MsgAddressInt
   value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
   created_lt:uint64 created_at:uint32 = CommonMsgInfo;
ext_in_msg_info$10 src:MsgAddressExt dest:MsgAddressInt
   import_fee:Grams = CommonMsgInfo;
ext_out_msg_info$11 src:MsgAddressInt dest:MsgAddressExt
   created_lt:uint64 created_at:uint32 = CommonMsgInfo;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
   code:(Maybe ^Cell) data:(Maybe ^Cell)
    library:(Maybe ^Cell) library:(Maybe ^Cell) = StateInit;
message$_ {X:Type} info:CommonMsgInfo
   init:(Maybe (Either StateInit ^StateInit))
   body:(Either X ^X) = Message X;
```

Смысл этой схемы заключается в следующем.

Тип Message X описывает сообщение с телом (или полезной нагрузкой) типа X. *Его сериализация начинается с информации типа*

CommonMsgInfo, которая поставляется в трех вариантах: для внутренних сообщений, входящих внешних сообщений и исходящих внешних сообщений соответственно. Все они имеют исходный адрес src и адрес назначения dest, которые являются внешними или внутренними в соответствии с выбранным конструктором. Кроме того, внутреннее сообщение может иметь некоторую ценность в граммах и других определенных валютах (ср. 3.1.6), и все сообщения, генерируемые внутри блокчейна TON, имеют логическое время создания created_lt (см. 1.4.6) и created_at создания unixtime, оба автоматически устанавливаются генерирующей транзакцией. Создание unixtime равно созданию unixtime блока, содержащего генерирующую транзакцию.

3.1.8. Экспедиторские сборы и сборы за ММСП. Общая стоимость внутреннего сообщения. Внутренние сообщения определяют ihr_fee в граммах, которое вычитается из значения, придаваемого сообщению, и присуждается валидаторам целевой цепочки, если они включают сообщение механизмом ММСП. Fwd_fee является первоначальная общая экспедиторская плата, уплаченная с использованием механизма HR; он автоматически вычисляется из некоторых параметров конфигурации и размера сообщения на момент создания сообщения.

Обратите внимание, что общее значение, переносимое вновь созданным внутренним исходящим сообщением, равно сумме значений, ihr_fee и fwd_fee. Эта сумма вычитается из остатка на счете источника. Из этих компонентов только значение всегда зачисляется на целевой счет при доставке сообщений. Fwd_fee собирается валидаторами на пути HR от источника до места назначения, а ihr_fee либо собирается валидаторами целевого шардчейна (если сообщение доставляется через ММСП), либо зачисляется на целевой счет.

3.1.9. Части кода и данных, содержащиеся в сообщении. Помимо общей информации о сообщении, хранящейся в информации, сообщение может содержать части кода и данных смарт-контракта назначения. Эта функция используется, например, в так называемых *сообщениях конструктора* (см. 1.7.3), которые являются просто внутренними или входящими внешними сообщениями с кодом и, возможно, полями данных, определенными в их инициализированных частях. Если хэш этих полей правильный, а конечный

смарт-контракт не имеет кода или данных, вместо него используются значения из сообщения.[24]

3.1.10. Использование кода и данных для других целей. Рабочие цепочки, отличные от мастерчейна и основной рабочей цепи, могут свободно использовать деревья ячеек, упомянутые в коде, данных и полях библиотеки, для своих собственных целей. Сама система обмена сообщениями не делает никаких предположений относительно их содержания; они становятся актуальными только тогда, когда сообщение обрабатывается транзакцией.

3.1.11. Отсутствие явной цены на газ и лимита газа. Обратите внимание, что в сообщениях нет четкой цены на газ и лимита газа. Вместо этого цена на газ устанавливается глобально валидаторами для каждой рабочей цепи (это специальный настраиваемый параметр), а лимит газа для каждой транзакции также имеет значение по умолчанию, которое является настраиваемым параметром; сам смарт-контракт может снизить лимит газа во время его исполнения, если это необходимо.

Для внутренних сообщений начальный лимит газа не может превышать значение Gram сообщения, деленное на текущую цену газа. Для входящих внешних сообщений начальный предел газа очень мал, а истинный предел газа устанавливается самим принимающим смарт-контрактом, когда он *принимает* входящее сообщение соответствующим примитивом TVM.

3.1.12. Десериализация полезной нагрузки сообщения. Полезная нагрузка или тело сообщения десериализуется принимающим смарт-контрактом при выполнении TVM. Сама система обмена сообщениями не делает никаких предположений о внутреннем формате полезной нагрузки. Однако имеет смысл описать сериализацию поддерживаемых входящих сообщений по схемам TL или TL-B с 32-битными тегами конструктора, чтобы

[24] Точнее, информация из поля init входящего сообщения используется либо тогда, когда принимающая учетная запись не инициализирована или заморожена с хэшем *StateInit,* равным ожидаемому учетной записью, либо когда принимающая учетная запись активна, а ее код или данные являются внешней хэш-ссылкой, соответствующей хэшу кода или данным, полученным в *StateInit* сообщения.

разработчики других смарт-контрактов знали интерфейс, поддерживаемый конкретным смарт-контрактом.

Сообщение всегда сериализуется внутри блокчейна как последнее поле в ячейке. Таким образом, программное обеспечение блокчейна может предположить, что любые биты и ссылки, оставшиеся неразобранными после разбора полей Message, предшествующего body, принадлежат к полезной нагрузке body : *X, ничего не зная о сериализации типа X.*

3.1.13. Сообщения с пустыми полезными нагрузками. Полезная нагрузка сообщения может оказаться фрагментом пустой ячейки, не имеющим битов данных и ссылок. По соглашению, такие сообщения используются для простой передачи ценностей. Обычно ожидается, что принимающий смарт-контракт будет обрабатывать такие сообщения тихо и успешно завершать работу (с нулевым кодом выхода), хотя некоторые смарт-контракты могут выполнять нетривиальные действия даже при получении сообщения с пустой полезной нагрузкой. Например, смарт-контракт может проверить полученный баланс, и, если его станет достаточно для ранее отложенного действия, запустить это действие. В качестве альтернативы, смарт-контракт может захотеть запомнить в своем постоянном хранилище полученную сумму и соответствующего отправителя, чтобы, например, распределить некоторые токены позже каждому отправителю пропорционально переведенным средствам.

Обратите внимание, что даже если смарт-контракт не делает специальных положений для сообщений с пустой полезной нагрузкой и выдает исключение при обработке таких сообщений, полученное значение (за вычетом платы за газ) все равно будет добавлено к балансу смарт-контракта.

3.1.14. Адрес источника сообщения и время логического создания определяют его генерирующий блок. Обратите внимание, что *исходный адрес и время логического создания внутреннего или исходящего внешнего сообщения однозначно определяют блок, в котором было создано сообщение. Действительно, адрес источника определяет исходный шардчейн, а блокам этого шардчейна присваиваются непересекающиеся логические временные интервалы, поэтому только один из них может содержать указанное время логического создания.*

Именно по этой причине в сообщениях не требуется явного упоминания о генерирующем блоке.

3.1.15. Сообщения в конвертах. *Конверты сообщений* используются для прикрепления информации о маршрутизации, такой как текущий (транзитный) адрес и адрес следующего прыжка, к входящим, транзитным и исходящим сообщениям (см. 2.1.16). Само сообщение хранится в отдельной ячейке и ссылается из конверта сообщения ссылкой на ячейку.

```
interm_addr_regular$0 use_src_bits:(#<= 96)
  = IntermediateAddress;
interm_addr_simple$10 workchain_id:int8 addr_pfx:(64 * Bit)
  = IntermediateAddress;
interm_addr_ext$11 workchain_id:int32 addr_pfx:(64 * Bit)
  = IntermediateAddress;
msg_envelope cur_addr: IntermediateAddress
  next_addr: IntermediateAddress fwd_fee_remaining:Grams
  msg:^(Message Any) = MsgEnvelope;
```

Тип IntermediateAddress используется для описания промежуточных адресов сообщения, то есть его текущего (или транзитного) адреса cur_addr и адреса следующего прыжка next_addr. Первый конструктор interm_addr_regular представляет промежуточный адрес с использованием оптимизации, описанной в 2.1.15, путем сохранения количества первых битов промежуточного адреса, которые совпадают с исходным адресом; два других явно хранят идентификатор рабочей цепочки и первые 64 бита адреса внутри этой рабочей цепи (остальные биты могут быть взяты из исходного адреса). Поле fwd_fee_remaining используется для явного представления максимальной суммы платы за пересылку сообщений, которая может быть вычтена из значения сообщения на оставшихся этапах управления персоналом; он не может превышать значение fwd_fee, указанное в самом сообщении.

3.2 Дескрипторы входящих сообщений

В этом разделе обсуждается *InMsgDescr, структура, содержащая описание всех входящих сообщений, импортированных в блок.*[25]

3.2.1. Типы и источники входящих сообщений. Каждое входящее сообщение, упомянутое в *InMsgDescr*, описывается значением типа *InMsg* («дескриптор входящего сообщения»), в котором указывается источник сообщения, причина его импорта в этот блок и некоторая информация о его «судьбе» — его обработка транзакцией или пересылка внутри блока. Входящие сообщения можно классифицировать следующим образом:

- *Входящие внешние сообщения* — не нуждаются в дополнительной причине для импорта в блок, но должны быть немедленно обработаны транзакцией в том же блоке.

- *Внутренние сообщения ММСП с адресами назначения в этом блоке* — Причина их импорта в блок включает в себя доказательство Меркла их генерации (т. е. их включение в *OutMsgDescr* их исходного блока). Такое сообщение должно быть немедленно доставлено в конечный пункт назначения и обработано транзакцией.

- *Внутренние сообщения с местами назначения в этом блоке* — Причиной их включения является их присутствие в *OutMsgQueue* самого последнего состояния соседнего шардчейна [26] или их присутствие в *OutMsgDescr* этого самого блока. Этот соседний шардчейн полностью определяется транзитным адресом, указанным в конверте пересылаемого сообщения, который также реплицируется в *InMsg*. «Судьба» этого сообщения снова описывается ссылкой на транзакцию обработки внутри текущего блока.

- *Немедленно маршрутизируемые внутренние сообщения* — по существу подкласс предыдущего класса сообщений. В этом случае

[25] Строго говоря, *InMsgDescr* является *типом* этой структуры; мы намеренно используем одну и ту же нотацию для описания единственного экземпляра этого типа в блоке.

[26] Напомним, что шардчейн считается соседом самого себя.

импортированное сообщение является одним из исходящих сообщений, сгенерированных в этом самом блоке.

- *Транзитные внутренние сообщения* — имеют ту же причину включения, что и предыдущий класс сообщений. Однако они не обрабатываются внутри блока, а внутренне перенаправляются в *OutMsgDescr* и *OutMsgQueue. Этот факт, наряду со ссылкой на новый конверт транзитного сообщения, должен быть зарегистрирован в InMsg.*

- *Отклоненные внутренние сообщения с местами назначения в этом блоке* — Внутреннее сообщение с назначением в этом блоке может быть импортировано и немедленно отброшено вместо обработки транзакцией, если оно уже было получено и обработано через ММСП в предыдущем блоке этого шардчейна. В этом случае должна быть предоставлена ссылка на предыдущую транзакцию обработки.

- *Отклоненные транзитные внутренние сообщения* — Аналогичным образом, транзитное сообщение может быть отброшено сразу после импорта, если оно уже было доставлено с помощью ММСП в конечный пункт назначения. В этом случае требуется доказательство Меркла о его обработке в конечном блоке (в виде сообщения ММСП).

3.2.2. Дескриптор входящего сообщения. Каждое входящее сообщение описывается экземпляром типа InMsg, который имеет шесть конструкторов, соответствующих случаям, перечисленным выше в 3.2.1:

```
msg_import_ext$000 msg:^( Message Any) transaction:^ Transaction
                   = InMsg;
msg_import_ihr$010 msg:^(Message Any) transaction:^ Transaction
    ihr_fee:Grams proof_created:^Cell = InMsg;
msg_import_imm$011 in_msg:^MsgEnvelope
    transaction:^ Transaction fwd_fee:Grams = InMsg;
msg_import_fin$100 in_msg:^MsgEnvelope
    transaction:^ Transaction fwd_fee:Grams = InMsg;
```

```
msg_import_tr$101 in_msg:^MsgEnvelope out_msg:^MsgEnvelope
    transit_fee:Grams = InMsg;
msg_discard_fin$110 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams = InMsg;
msg_discard_tr$111 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams proof_delivered:^Cell = InMsg;
```

Обратите внимание, что транзакция обработки ссылается в первых четырех конструкторах непосредственно на ячейку, ссылающуюся на Транзакцию, хотя для этой цели достаточно логического времени транзакции transaction_lt:uint64. Внутренние условия согласованности гарантируют, что транзакция, о которой идет речь, действительно принадлежит смарт-контракту назначения, указанному в сообщении, и что входящее сообщение, обработанное этой транзакцией, действительно описывается в данном *экземпляре InMsg.*

Кроме того, обратите внимание, что msg_import_imm можно отличить от msg_import_fin, заметив, что это единственный случай, когда время логического создания обрабатываемого сообщения больше или равно (минимальному) логическому времени блока, импортирующего сообщение.

3.2.3. Взимание экспедиторских и транзитных сборов с ввозимых сообщений. Структура *InMsg* также используется для указания платы за пересылку и транзит, взимаемую с входящих сообщений. Сама плата указывается в полях ihr_fee, fwd_fee или transit_fee; он отсутствует только во входящих внешних сообщениях, которые используют другие механизмы для вознаграждения валидаторов за их импорт. Сборы должны удовлетворять следующим внутренним условиям согласованности:

- Для внешних сообщений (msg_import_ext) плата за пересылку не взимается.
- Для импортированных ММСП внутренних сообщений (msg_import_ihr) плата равна ihr_fee, которая должна совпадать со значением ihr_fee, указанным в самом сообщении. Обратите внимание, что fwd_fee или fwd_fee_remaining никогда не собираются из сообщений, импортированных ММСП.

- За внутренние сообщения, доставленные в пункт назначения (msg_import_fin и msg_import_imm), плата равна fwd_fee_remaining in_msg входящих сообщений в оболочке. Обратите внимание, что он не может превышать значение fwd_fee, указанное в самом сообщении.

- Для транзитных сообщений (msg_import_tr) плата равна разнице между fwd_fee_remaining значениями, указанными в конвертах in_msg и out_msg.

- За отклоненные сообщения плата также равна fwd_fee_remaining, указанной в in_msg.

3.2.4. Импортированное значение входящего сообщения. Каждое импортированное сообщение импортирует в блок некоторое значение — определенное количество одной или нескольких криптовалют. Это импортированное значение вычисляется следующим образом:

- Внешнее сообщение не импортирует значение.

- Импортированное ММСП сообщение импортирует свою стоимость плюс ihr_fee.

- Доставленное или транзитное внутреннее сообщение импортирует свою стоимость плюс ihr_fee плюс стоимость fwd_fee_remaining своего in_msg конверта.

- Отброшенное сообщение импортирует fwd_fee_remaining своего in_msg конверта.

Обратите внимание, что сборы за пересылку и транзит, взимаемые с импортированного сообщения, не превышают его импортированную стоимость.

3.2.5. Дополненные хэш-карты, или словари. Прежде чем продолжить, давайте обсудим сериализацию дополненных хэш-карт, или словарей.

Дополненные хэш-карты представляют собой структуры хранения ключ-значение с *n-битными ключами и значениями некоторого типа X, аналогичные обычным хэш-картам, описанным в [4, 3.3]. Однако каждый*

промежуточный узел дерева Патрисии, представляющий дополненную хэш-карту, дополняется значением типа Y.

Эти значения увеличения должны удовлетворять определенным *условиям агрегирования. Как правило,* Y является целочисленным типом, и условием агрегирования является то, что значение дополнения форка должно равняться сумме значений аугментации двух его дочерних элементов. В общем, вместо *суммы используется функция оценки форка* $S : Y \times Y \to Y$ или $S : Y \to Y \to Y$. Значение увеличения листа обычно вычисляется из значения, хранящегося в этом листе, с помощью функции *оценки листа* $L : X \to Y$. Значение увеличения листа может быть явно сохранено в листе вместе со значением; однако в большинстве случаев в этом нет необходимости, поскольку функция оценки листа L очень проста.

3.2.6. Сериализация дополненных хэш-карт. Сериализация дополненных хэш-карт с *n-битными ключами, значениями типа* X *и значениями аугментации типа* Y задается следующей схемой TL-B, которая является расширением схемы, приведенной в [4, 3.3.3]:

```
ahm_edge#_ {n:#} {X:Type} {Y:Type} {l:#} {m:#}
      label:(HmLabel ~l n) {n = (~m) + l}
      node:(HashmapAugNode m X Y) = HashmapAug n X Y;
ahmn_leaf#_ {X:Type} {Y:Type} extra:Y value:X
   = HashmapAugNode 0 X Y;
ahmn_fork#_ {n:#} {X:Type} {Y:Type}
      left:^(HashmapAug n X Y) right:^(HashmapAug n X Y) extra:Y
   = HashmapAugNode (n + 1) X Y;

ahme_empty$0 {n:#} {X:Type} {Y:Type} extra:Y
                = HashmapAugE n X Y;
ahme_root$1 {n:#} {X:Type} {Y:Type} root:^(HashmapAug n X Y)
    extra:Y = HashmapAugE n X Y;
```

3.2.7. Аугментация *InMsgDescr. Коллекция дескрипторов входящих сообщений дополняется вектором из двух валютных значений,*

представляющих импортированную стоимость и сборы за пересылку и транзит, взимаемые с сообщения или коллекции сообщений:

```
import_fees$_ fees_collected:Grams
    value_imported:CurrencyColection = ImportFees;
```

3.2.8. Структура *InMsgDescr. Теперь сам InMsgDescr* определяется как дополненная хэш-карта, с 256-битными ключами (равными хэшам представления импортированных сообщений), значениями типа InMsg (ср. 3.2.2) и значениями дополнения типа ImportFees (ср. 3.2.7):

```
_ (HashmapAugE 256 InMsg ImportFees) = InMsgDescr;
```

Эта нотация TL-B использует анонимный конструктор _ для определения InMsgDescr как синонима другого типа.

3.2.9. Правила агрегации для *InMsgDescr. Функции оценки вилки и оценки листа (см. 3.2.5) явно не включены в приведенную выше нотацию, поскольку зависимые типы TL-B недостаточно выразительны для этой цели. Другими словами, функция оценки форка — это просто компонентное добавление двух экземпляров ImportFees, а функция конечной оценки определяется правилами, перечисленными в разделах 3.2.3 и 3.2.4. Таким образом, корень дерева Patricia, представляющий экземпляр InMsgDescr,* содержит экземпляр *ImportFees* с общим значением, импортированным всеми входящими сообщениями, и с общей платой за пересылку, взимаемой с них.

3.3 Очередь исходящих сообщений и дескрипторы

В этом разделе обсуждается *OutMsgDescr, структура, представляющая все исходящие сообщения блока, а также их конверты и краткие описания причин их включения в OutMsgDescr. Эта структура также описывает все модификации OutMsgQueue, который является частью состояния shardchain.*

3.3.1. Типы исходящих сообщений. Исходящие сообщения можно классифицировать следующим образом:

- *Внешние исходящие сообщения, или «сообщения в никуда» — генерируются транзакцией внутри этого блока. Причиной включения такого сообщения в OutMsgDescr* является просто ссылка на его генерирующую транзакцию.

- *Немедленно обрабатываются внутренние исходящие сообщения* — генерируются и обрабатываются именно в этом блоке, а не включаются в *OutMsgQueue. Причиной включения такого сообщения является ссылка на его генерирующую транзакцию, а его «судьба» описывается ссылкой на соответствующую запись в InMsgDescr.*

- *Обычные (внутренние) исходящие сообщения* — генерируются в этом блоке и включаются в *OutMsgQueue.*

- *Транзитные (внутренние) исходящие сообщения* — импортируются в *InMsgDescr* того же блока и маршрутизируются через HR до тех пор, пока не будет получен адрес следующего прыжка за пределами текущего сегмента.

3.3.2. Записи о списании сообщений. Помимо вышеуказанных типов исходящих сообщений, *OutMsgDescr* может содержать специальные «записи о списании сообщений», которые указывают на то, что сообщение было удалено из *OutMsgQueue* в этом блоке. Причина этого удаления указана в записи об удалении сообщения; он состоит из ссылки на удаляемое сообщение в оболочке и на логическое время соседнего блока

shardchain, в котором это оболваненное сообщение находится в его *InMsgDescr.*

Обратите внимание, что в некоторых случаях сообщение может быть импортировано из *OutMsgQueue* текущей цепочки сегментов, внутренне маршрутизировано, а затем снова включено в *OutMsgDescr* и *OutMsgQueue* с другим конвертом. [27] В этом случае используется вариант описания транзитного исходящего сообщения, который удваивается как запись вывода сообщения из очереди.

3.3.3. Дескриптор исходящего сообщения. Каждое исходящее сообщение описывается экземпляром *OutMsg:*

```
msg_export_ext$000 msg:^(Message Any)
        transaction:^ Transaction = OutMsg;
msg_export_imm$010 out_msg:^MsgEnvelope
        transaction:^Transaction reimport:^InMsg = OutMsg;
msg_export_new$001 out_msg:^MsgEnvelope
        transaction:^ Transaction = OutMsg;
msg_export_tr$011 out_msg:^MsgEnvelope
        imported:^InMsg = OutMsg;
msg_export_deq$110 out_msg:^MsgEnvelope
        import_block_lt:uint64 = OutMsg;
msg_export_tr_req$111 out_msg:^MsgEnvelope
        imported:^InMsg = OutMsg;
```

Последние два описания приводят к удалению (снятию) сообщения из *OutMsgQueue* вместо его вставки. Последний повторно вставляет сообщение в *OutMsgQueue* с новым конвертом после выполнения внутренней маршрутизации (см. 2.1.11).

[27] Такая ситуация встречается редко и возникает только после событий слияния шардчейнов. Обычно сообщения, импортированные из *OutMsgQueue* того же shardchain, имеют места назначения внутри этого shardchain и обрабатываются соответствующим образом, а не повторно ставятся в очередь.

3.3.4. Экспортированное значение исходящего сообщения. Каждое исходящее сообщение, описанное *OutMsg,* экспортирует некоторую ценность — определенное количество одной или нескольких криптовалют — из блока. Это экспортированное значение вычисляется следующим образом:

- Внешнее исходящее сообщение не экспортирует значения.
- Внутреннее сообщение, генерируемое в этом блоке, экспортирует свою стоимость плюс ihr_fee плюс fwd_fee. Обратите внимание, что fwd_fee должен быть равен fwd_fee_remaining, указанному в out_msg конверте.
- Транзитное сообщение экспортирует свою стоимость плюс ihr_fee плюс стоимость fwd_fee_remaining своего out_msg конверта.
- То же самое относится и к msg_export_tr_req, конструктору *OutMsg,* используемому для повторно вставленных сообщений, освобожденных от очереди.
- Запись о выселении сообщения (msg_export_deq; см. 3.3.2) не экспортирует ценность.

3.3.5. Структура *OutMsgDescr. Сам OutMsgDescr* представляет собой просто дополненную хэш-карту (ср. 3.2.5), с 256-битными ключами (равными хэшу представления сообщения), значениями типа *OutMsg и значениями дополнения типа CurrencyCollection:*

_ (HashmapAugE 256 OutMsg CurrencyCollection) = OutMsgDescr;

Увеличение представляет собой *экспортируемую стоимость* соответствующего сообщения, агрегированную с помощью суммы и рассчитанную на листьях, как описано в разделе 3.3.4. Таким образом, общая экспортируемая стоимость отображается рядом с корнем дерева Патрисии, представляющего *OutMsgDescr.*

Наиболее важным условием согласованности для *OutMsgDescr* является то, что его запись с ключом k должна быть *OutMsg,* описывающей сообщение m с хэшем представления $\text{Hash}^b(m) = k$.

3.3.6. Структура *OutMsgQueue. Напомним (ср. 1.2.7), что OutMsgQueue* является частью состояния блокчейна, а не блока. Поэтому блок содержит только хэш-ссылки на начальное и конечное состояние, а также вновь созданные ячейки.

Структура *OutMsgQueue* проста: это просто дополненная хэш-карта с 352-битными ключами и значениями типа *OutMsg:*

_ (HashmapAugE 352 OutMsg uint64) = OutMsgQueue;

Ключом, используемым для исходящего сообщения m, является объединение его 32-разрядного workchain_id следующего прыжка*, первых 64 бит адреса следующего прыжка внутри этой рабочей цепочки и хэша представления Hash[(m) самого* сообщения m.

Аугментация происходит по логическому времени создания Lt(m) сообщения m на листьях и по минимуму значений аугментации детей на развилках.

Наиболее важным условием согласованности для *OutMsgQueue* является то, что значение в ключе k действительно должно содержать сообщение в оболочке с ожидаемым адресом следующего прыжка и хэшем представления.

3.3.7. Условия согласованности для *OutMsg. Несколько внутренних условий согласованности накладываются на экземпляры OutMsg,* присутствующие в *OutMsgDescr. К ним относятся следующие:*

- Каждый из первых трех конструкторов описаний исходящих сообщений содержит ссылку на генерирующую транзакцию. Эта транзакция должна принадлежать исходной учетной записи сообщения, она должна содержать ссылку на указанное сообщение в качестве одного из исходящих сообщений и должна быть восстановлена путем поиска ее по account_id и transaction_id.

- msg_export_tr и msg_export_tr_req должны ссылаться на *экземпляр InMsg*, описывающий одно и то же сообщение (в другом исходном конверте).

- Если используется один из первых четырех конструкторов, сообщение должно отсутствовать в начальном *OutMsgQueue* блока; в противном случае оно должно присутствовать.
- Если используется msg_export_deq, сообщение должно отсутствовать в окончательном *OutMsgQueue* блока; в противном случае оно должно присутствовать.
- Если сообщение не упоминается в *OutMsgDescr, оно должно совпадать в начальном и конечном OutMsgQueues блока.*

4 Счета и операции

В этой главе обсуждается расположение учетных записей (или *смарт-контрактов) и их состояние* в блокчейне TON. Он также рассматривает *транзакции, которые являются единственным способом изменения состояния учетной записи, а также обработки входящих сообщений и создания новых исходящих сообщений.*

4.1 Счета и их состояния

Напомним, что *смарт-контракт* и учетная *запись* — это одно и то же в контексте блокчейна TON, и что эти термины могут использоваться взаимозаменяемо, по крайней мере, до тех пор, пока рассматриваются только небольшие (или «обычные») смарт-контракты. Большой смарт-контракт может использовать несколько учетных записей, лежащих в разных цепочках одной и той же рабочей цепи для целей балансировки нагрузки.

Учетная запись идентифицируется по ее полному адресу и *полностью описывается* ее состоянием. Другими словами, в аккаунте нет ничего другого, кроме его адреса и состояния.

4.1.1. Адреса учетных записей. Как правило, учетная запись полностью идентифицируется по ее *полному адресу, состоящему из 32-разрядной workchain_id, и (обычно 256-битного) внутреннего адреса* или *идентификатора учетной записи, account_id* внутри выбранной рабочей цепочки. В базовой рабочей цепочке (workchain_id = 0) и в мастерчейне (workchain_id = −1) внутренний адрес всегда 256-битный. В этих рабочих цепочках [28] *account_id* не может быть выбран произвольно, но должен быть равен хэшу исходного кода и данных смарт-контракта; в противном случае будет невозможно инициализировать счет предполагаемым кодом и данными (см. 1.7.3), а также что-либо сделать с накопленными средствами на балансе счета.

[28] Для простоты мы иногда рассматриваем мастерчейн как просто еще одну рабочую цепь с *workchain_id* = −1.

4.1.2. Нулевой счет. По соглашению, *нулевой счет* или *счет с нулевым адресом* накапливает сборы за обработку, пересылку и транзит, а также любые другие платежи, собираемые валидаторами мастерчейна или рабочей цепи. Кроме того, нулевой счет является «большим смарт-контрактом», что означает, что каждый шардчейн имеет свой экземпляр нулевой учетной записи, причем наиболее значимые биты адреса скорректированы, чтобы лежать в сегменте. Любые средства, переведенные на нулевой счет, намеренно или случайно, фактически являются подарком для валидаторов. Например, смарт-контракт может уничтожить себя, отправив все свои средства на нулевой счет.

4.1.3. Малые и большие смарт-контракты. По умолчанию смарт-контракты являются «маленькими», что означает, что у них есть один адрес учетной записи, принадлежащий ровно одному шардчейну в любой момент времени. Тем не менее, можно создать «большой смарт-контракт с глубиной разделения d», что означает, что может быть создано до 2D-экземпляров смарт-контракта, причем первые d биты исходного адреса смарт-контракта заменяются произвольными битовыми последовательностями. [29] На такие смарт-контракты можно отправлять сообщения, используя внутренние адреса anycast с любой трансляцией, установленной в d (см. 3.1.2). Кроме того, экземплярам большого смарт-контракта разрешено использовать этот адрес anycast в качестве исходного адреса своих сгенерированных сообщений.

Экземпляром большого смарт-контракта является счет с ненулевой *максимальной глубиной расщепления d.*

4.1.4. Три вида счетов. Существует три типа учетных записей:

- *Неинициализированный* — счет имеет только баланс; его код и данные еще не инициализированы.

[29] Фактически, до первых d биты заменяются таким образом, что каждый сегмент содержит не более одного экземпляра большого смарт-контракта, и что сегменты (w,s) с префиксом s длины $|s| \le d$ содержат ровно один экземпляр.

- *Активный* — код и данные учетной записи также были инициализированы.
- *Заморожено* — код и данные учетной записи были заменены хэшем, но баланс по-прежнему хранится явно. Остаток на замороженном счете может фактически стать отрицательным, отражая причитающиеся платежи за хранение.

4.1.5. Профиль хранения учетной записи. Профиль *хранения* учетной записи — это структура данных, описывающая объем постоянного хранилища состояния блокчейна, используемого этой учетной записью. Он описывает общее количество используемых ячеек, битов данных, а также внутренних и внешних ссылок на ячейки.

```
storage_used$_ cells:(VarUInteger 7) bits:(VarUInteger 7)
  ext_refs:(VarUInteger 7) int_refs:(VarUInteger 7)
  public_cells:(VarUInteger 7) = StorageUsed;
```

Тот же тип StorageUsed может представлять профиль хранения сообщения, как это требуется, например, для вычисления fwd_fee, общей платы за пересылку для маршрутизации Hypercube. Профиль хранения учетной записи имеет несколько дополнительных полей, указывающих на последний раз, когда взималась плата за хранение:

```
storage_info$_ used:StorageUsed last_paid:uint32
                  due_payment:(Maybe Grams) = StorageInfo;
```

Поле last_paid содержит либо unixtime последнего собранного платежа за хранилище (обычно это unixtime самой последней транзакции), либо unixtime, когда была создана учетная запись (опять же, транзакцией). Поле due_payment, если оно присутствует, аккумулирует платежи за хранение, которые не могли быть взысканы с баланса счета, представленные строго положительным количеством наноГраммов; он может присутствовать только для неинициализированных или замороженных счетов, которые имеют баланс ноль граммов (но могут иметь ненулевые балансы в других криптовалютах). Когда due_payment становится больше значения настраиваемого параметра блокчейна, аккаунт полностью уничтожается, а его баланс, если таковой имеется, переводится на нулевой счет.

4.1.6. Описание учетной записи. Состояние учетной записи представлено экземпляром типа *Account, описанным следующей схемой TL-B:*[30]

```
account_none$0 = Account;
account$1 addr:MsgAddressInt storage_stat:StorageInfo
            storage:AccountStorage = Account;

account_storage$_ last_trans_lt:uint64
    balance:CurrencyCollection state:AccountState
  = AccountStorage;

account_uninit$00 = AccountState;
account_active$1 _:StateInit = AccountState;
account_frozen$01 state_hash:uint256 = AccountState;

acc_state_uninit$00 = AccountStatus;
acc_state_frozen$01 = AccountStatus;
acc_state_active$10 = AccountStatus;
acc_state_nonexist$11 = AccountStatus;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell) library:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

Обратите внимание, что account_frozen содержит хэш представления экземпляра *StateInit, а не самого этого экземпляра, который в противном случае содержался бы в account_active; account_uninit похож на account_frozen, но не содержит явного state_hash, поскольку*

[30] В этой схеме используются анонимные конструкторы и анонимные поля, представленные символом подчеркивания _.

предполагается, что он равен внутреннему адресу учетной записи (account_id), уже присутствует в поле addr. Поле split_depth присутствует и ненулевое только в случаях крупных смарт-контрактов. Специальное поле может присутствовать только в мастерчейне — и внутри мастерчейна, только в некоторых фундаментальных смарт-контрактах, необходимых для функционирования всей системы.

Статистика хранилища, хранящаяся в storage_stat отражает общее использование хранилища среза ячейки. В частности, биты и ячейки, используемые для хранения баланса, также отражаются в storage_stat.

Когда необходимо представить несуществующую учетную запись, используется конструктор account_none.

4.1.7. Состояние учетной записи как сообщение от учетной записи к ее будущему я. Обратите внимание, что состояние учетной записи очень похоже на сообщение, отправленное со счета его будущему участию в следующей транзакции, по следующим причинам:

- Состояние счета не меняется между двумя последовательными транзакциями одного и того же счета, поэтому оно полностью похоже в этом отношении на сообщение, отправленное с более ранней транзакции на более позднюю.

- Когда транзакция обрабатывается, ее входными данными являются входящее сообщение и предыдущее состояние учетной записи; его выходными данными являются генерируемые исходящие сообщения и состояние следующей учетной записи. Если рассматривать состояние как особый вид сообщения, то мы увидим, что каждая транзакция имеет ровно два входа (состояние учетной записи и входящее сообщение) и по крайней мере один выход.

- Как сообщение, так и состояние учетной записи могут содержать код и данные в экземпляре *StateInit, а также некоторое значение в их балансе.*

- Учетная запись инициализируется сообщением *конструктора, которое по существу несет будущее состояние и баланс счета.*

- В некоторых случаях сообщения преобразуются в состояния учета, и наоборот. Например, когда происходит событие слияния shardchain, и необходимо объединить две учетные записи, которые являются экземплярами одного и того же большого контракта, одна из них преобразуется в сообщение, отправленное другому (см. 4.2.11). Аналогично, когда происходит событие разделения shardchain, и экземпляр большого смарт-контракта должен быть разделен на две части, это достигается специальной транзакцией, которая создает новый экземпляр с помощью сообщения конструктора, отправленного из ранее существующего экземпляра в новый (см. 4.2.10).

- Можно сказать, что сообщение связано с передачей некоторой информации *через пространство* (между различными шардчейнами или, по крайней мере, связками счетов), в то время как состояние учетной записи передает информацию *во времени* (из прошлого в будущее одной и той же учетной записи).

4.1.8. Различия между сообщениями и состояниями аккаунта. Конечно, есть и важные отличия. Например:

- Состояние аккаунта передается только «во времени» (для блока шардчейна его преемнику), но никогда «в пространстве» (от одного шардчейна к другому). Как следствие, эта передача осуществляется неявно, без создания полных копий состояния учетной записи в любом месте блокчейна.

- Платежи за хранилище, собираемые валидаторами для поддержания состояния учетной записи, обычно значительно меньше, чем плата за пересылку сообщений для того же объема данных.

- Когда входящее сообщение доставляется в учетную запись, вызывается код из учетной записи, а не код из сообщения.

4.1.9. Совокупное состояние всех счетов в сегменте. Разделенная часть состояния шардчейна (см. 1.2.1 и 1.2.2) задается формулой

_ (HashmapAugE 256 Account CurrencyCollection) = ShardAccounts;

Это просто словарь с 256-битными *account_ids в качестве ключей и соответствующими состояниями счета в виде значений, увеличенных суммой остатков на счетах. Таким образом, вычисляется сумма остатков всех счетов в шардчейне, так что можно легко проверить общее количество криптовалюты, «хранящейся» в осколке.*

Внутренние условия согласованности гарантируют, что адрес учетной записи, на которую ссылается ключ k в *SmartAccounts,* действительно равен k. *Дополнительное условие внутренней согласованности требует, чтобы все клавиши* k начинались с префикса сегмента s.

4.1.10. Описание владельца аккаунта и интерфейса. Можно включить некоторую дополнительную информацию в контролируемый счет. Например, отдельный пользователь или компания могут захотеть добавить текстовое поле описания в свою учетную запись кошелька с именем или адресом пользователя или компании (или их хэшем, если информация не должна быть общедоступной). В качестве альтернативы, смарт-контракт может предлагать машиночитаемое или читаемое человеком описание поддерживаемых им методов и их предполагаемого применения, которое может использоваться продвинутыми приложениями кошелька для создания раскрывающихся меню и форм, помогающих пользователю-человеку создавать действительные сообщения для отправки на этот смарт-контракт.

Одним из способов включения такой информации является резервирование, скажем, второй ссылки в ячейке данных состояния счета для словаря с 64-битными ключами (соответствующими некоторым идентификаторам стандартных типов дополнительных данных, которые можно было бы захотеть сохранить) и соответствующими значениями. Тогда исследователь блокчейна сможет извлечь требуемое значение вместе с доказательством Меркла, если это необходимо.

Лучший способ сделать это – определить некоторые *методы получения* в смарт-контракте.

4.1.11. Получение методов смарт-контракта. *Методы Get* выполняются автономным экземпляром TVM с кодом учетной записи и загруженными в нее данными. Требуемые параметры передаются в стек (скажем, магическое число, указывающее на извлекаемое поле или конкретный

метод get, который необходимо вызвать), а результаты также возвращаются в стек TVM (скажем, фрагмент ячейки, содержащий сериализацию строки с именем владельца учетной записи).

В качестве бонуса методы get могут использоваться для получения ответов на более сложные запросы, чем просто получение постоянного объекта. Например, смарт-контракты реестра TON DNS предоставляют методы get для поиска строки домена в реестре и возврата соответствующей записи, если она найдена.

По соглашению, методы получения используют большие *отрицательные* 32-битные или 64-битные индексы или магические числа, а внутренние функции смарт-контракта используют последовательные положительные индексы, которые будут использоваться в инструкции TVM CALLDICT. Функция main() смарт-контракта, используемая для обработки входящих сообщений в обычных транзакциях, всегда имеет нулевой индекс.

4.2 Операций

Согласно парадигме бесконечного шардинга и модели актора, тремя основными компонентами блокчейна TON являются *учетные записи* (вместе с их состояниями), *сообщения* и *транзакции. В предыдущих разделах уже обсуждались первые два; в этом разделе рассматриваются транзакции.*

В отличие от сообщений, которые имеют по существу одинаковые заголовки во всех рабочих цепочках TON Blockchain, и учетных записей, которые имеют по крайней мере некоторые общие части (адрес и баланс), наше обсуждение транзакций обязательно ограничивается мастерчейном и основным рабочим цепью. Другие рабочие цепочки могут определять совершенно другие виды транзакций.

4.2.1. Логическое время транзакции. Каждой транзакции t присвоен логический временной интервал Lt•(t) = [Lt−(t),Lt+(t)) (ср. 1.4.6 и 1.4.3).

По соглашению, транзакции t, генерирующей n исходящих сообщений $m1,\ldots, mn$, присваивается логический интервал времени длиной $n + 1$, так что

$$\mathrm{Lt}+(t) = \mathrm{Lt}-(t) + n + 1 \quad . \tag{16}$$

Мы также устанавливаем Lt(t) := Lt−(t) и присваиваем время логического создания сообщения mi, *где* $1 \le i \le n$, *по*

$$\mathrm{Lt}(mi) = \mathrm{Lt}-(mi) := \mathrm{Lt}-(t) + i, Lt \qquad +(mi) := \mathrm{Lt}-(mi) + 1 \quad . \tag{17}$$

Таким образом, каждому сгенерированному исходящему сообщению присваивается свой собственный единичный интервал внутри логического временного интервала Lt•(t) транзакции t.

4.2.2. Логическое время однозначно идентифицирует транзакции и исходящие сообщения счета. Напомним, что условия, налагаемые на логическое время, подразумевают, что Lt−(t) ≥ Lt+(t') для любой предшествующей транзакции t' того же счета ξ, *и что* $Lt-(t) >$ Lt(m), если m является входящим сообщением, обрабатываемым транзакцией t. *Таким образом, логические временные* интервалы транзакций одного и того же счета не пересекаются друг с другом, и, как следствие, логические

временные интервалы всех исходящих сообщений, генерируемых учетной записью, также не пересекаются друг с другом. Другими словами, все Lt(m) различны, когда m проходит через все исходящие сообщения одной учетной записи ξ.

Таким образом, Lt(t) и Lt(m) в сочетании с идентификатором учетной записи ξ *однозначно определяют транзакцию* t или исходящее сообщение m этой учетной записи. Кроме того, если у вас есть упорядоченный список всех транзакций счета вместе с их логическим временем, легко найти транзакцию, которая сгенерировала данное исходящее сообщение m, *просто посмотрев транзакцию* t с логическим временем Lt(t), ближайшим к Lt(m) снизу.

4.2.3. Родовые компоненты транзакции. Каждая транзакция t содержит или косвенно ссылается на следующие данные:

- Счет ξ, к которому принадлежит транзакция.
- Логическое время Lt(t) транзакции.
- Одно или ноль входящих сообщений m обрабатывается транзакцией.
- Количество сгенерированных исходящих сообщений $n \geq 0$.
- Исходящие сообщения $m_1, \ldots, m_n$.
- Начальное состояние счета ξ (включая его баланс).
- Конечное состояние счета ξ (включая его баланс).
- Общая сумма сборов, взимаемых валидаторами.
- Подробное описание транзакции, содержащее все или некоторые данные, необходимые для ее проверки, включая вид транзакции (см. 4.2.4) и некоторые из выполненных промежуточных шагов.

Из этих компонентов все, кроме самого последнего, являются довольно общими и могут появляться и в других рабочих цепочках.

4.2.4. Виды сделок. Существуют различные виды транзакций, разрешенных в мастерчейне и шардчейнах. *Обычные транзакции* заключаются в доставке одного входящего сообщения на счет и его обработке кодом этого счета; это наиболее распространенный вид транзакции. Кроме того, существует несколько видов экзотических транзакций. Всего существует шесть видов сделок:

- *Обычные транзакции* — принадлежать счету ξ. *Они обрабатывают ровно одно входящее сообщение* m (описанное в *InMsgDescr* охватывающего блока) с назначением ξ, *вычисляют новое состояние учетной записи и генерируют несколько исходящих сообщений (зарегистрированных в OutMsgDescr) с исходным кодом* ξ.

- *Транзакции хранения* — могут быть вставлены валидаторами по своему усмотрению. Они не обрабатывают входящие сообщения и не вызывают код. Их единственным эффектом является сбор платежей за хранение со счета, влияя на его статистику хранения и его баланс. Если полученный Gram баланс счета становится меньше определенной суммы, счет может быть заморожен, а его код и данные заменены их комбинированным хэшем.

- *Тиковые транзакции* — автоматически вызываются для определенных специальных счетов (смарт-контрактов) в мастерчейне, в состоянии которых установлен флаг тика, как самые первые транзакции в каждом блоке мастерчейна. Они не имеют входящих сообщений, но могут генерировать исходящие сообщения и изменять состояние учетной записи. Например, *выборы валидатора* выполняются путем тиковых транзакций специальных смарт-контрактов в мастерчейне.

- *Транзакции Tock* — аналогично автоматически вызываются как самые последние транзакции в каждом блоке мастерчейна для определенных специальных учетных записей.

- *Разделенные транзакции* — вызываются как последние транзакции блоков shardchain, непосредственно предшествующие событию

разделения shardchain. Они запускаются автоматически для экземпляров крупных смарт-контрактов, которые должны создать новый экземпляр после разделения.

- *Транзакции слияния* — аналогично вызываются как первые транзакции блоков shardchain сразу после события слияния shardchain, если экземпляр большого смарт-контракта необходимо объединить с другим экземпляром того же смарт-контракта.

Обратите внимание, что из этих шести видов транзакций только четыре могут происходить в мастерчейне, а еще одно подмножество из четырех может происходить в основной рабочей цепочке.

4.2.5. Этапы обычной сделки. Обычная транзакция выполняется в несколько *этапов, которые можно рассматривать как несколько «субтранзакций», тесно связанных в одну:*

- *Этап хранения* — собирает причитающиеся платежи за состояние учетной записи (включая код смарт-контракта и данные, если таковые имеются) до настоящего времени. В результате смарт-контракт может быть *заморожен.* Если смарт-контракт не существовал раньше, фаза хранения отсутствует.

- *Кредитный этап* — на счет зачисляется значение полученного входящего сообщения.

- *Этап вычисления* — код смарт-контракта вызывается внутри экземпляра TVM с адекватными параметрами, включая копию входящего сообщения и постоянных данных, и завершается кодом выхода, новыми постоянными данными и списком действий (который включает, например, исходящие сообщения для отправки). Этап обработки может привести к созданию новой учетной записи (неинициализированной или активной) или к активации ранее неинициализированной или замороженной учетной записи. Плата *за газ, равная произведению цены на газ и потребленного газа, взимается с остатка на счете.*

- *Фаза действия* — если смарт-контракт успешно расторгнут (с кодом выхода 0 или 1), выполняются действия из списка. Если невозможно выполнить их все — например, из-за недостаточности средств для перевода с исходящим сообщением — то транзакция прерывается, а состояние счета откатывается. Транзакция также прерывается, если смарт-контракт не был успешно расторгнут, или если не было возможности вызвать смарт-контракт вообще, потому что он не инициализирован или заморожен.

- *Фаза отказа* — если транзакция была прервана, а для входящего сообщения установлен флаг bounce, то оно «отскакивает» путем автоматического создания исходящего сообщения (с четким флагом bounce) его первоначальному отправителю. Почти вся стоимость исходного входящего сообщения (за вычетом платежей за газ и платы за пересылку) передается в сгенерированное сообщение, которое в противном случае имеет пустой текст.

4.2.6. Передача входящих сообщений на несуществующие учетные записи. Обратите внимание, что если входящее сообщение с установленным флагом bounce отправляется на ранее несуществующую учетную запись, а транзакция прерывается (например, из-за того, что во входящем сообщении нет кода и данных с правильным хэшем, поэтому виртуальная машина вообще не может быть вызвана), то учетная запись не создается даже как неинициализированная учетная запись, так как он будет иметь нулевой баланс и никакого кода и данных в любом случае.[31]

4.2.7. Обработка входящего сообщения разделена между этапами вычислений и действий. Обратите внимание, что обработка входящего сообщения фактически разделена на две фазы: фазу *вычисления* и фазу *действия.* На этапе вычисления вызывается виртуальная машина и

[31] В частности, если пользователь по ошибке отправит некоторые средства на несуществующий адрес в отказываемом сообщении, средства не будут потрачены впустую, а скорее будут возвращены (возвращены) обратно. Поэтому приложение кошелька пользователя должно устанавливать флаг возврата во всех сгенерированных сообщениях по умолчанию, если явно не указано иное. Однако в некоторых ситуациях сообщения, не подлежащие отражению, необходимы (см. 1.7.6).

выполняются необходимые вычисления, но никакие действия за пределами виртуальной машины не выполняются. Другими словами, *исполнение смарт-контракта в TVM не имеет побочных эффектов; у смарт-контракта нет возможности взаимодействовать с блокчейном непосредственно во время его исполнения. Вместо этого tvM-примитивы, такие как SENDMSG, просто сохраняют требуемое действие (например, исходящее сообщение, которое должно быть отправлено) в списке действий, постепенно накапливаемом в управляющем регистре TVM c5. Сами действия откладываются до фазы действия, во время которой смарт-контракт пользователя вообще не вызывается.*

4.2.8. Причины разделения обработки на этапы вычислений и действий. Некоторые причины такой договоренности:

- Проще прервать транзакцию, если смарт-контракт в конечном итоге прекращается с кодом выхода, отличным от 0 или 1.
- Правила обработки выходных действий могут быть изменены без изменения виртуальной машины. (Например, могут быть введены новые выходные действия.)
- Сама виртуальная машина может быть модифицирована или даже заменена другой (например, в новой рабочей цепочке) без изменения правил обработки выходных действий.
- Выполнение смарт-контракта внутри виртуальной машины полностью изолировано от блокчейна и является *чистым вычислением. Как следствие, это выполнение может быть виртуализировано* внутри самой виртуальной машины с помощью примитива RUNVM от TVM, полезной функции для смарт-контрактов валидатора и для смарт-контрактов, контролирующих платежные каналы и другие сайдчейны. Кроме того, виртуальная машина может *эмулироваться* внутри себя

или урезанной версией себя, что является полезной функцией для проверки выполнения смарт-контрактов внутри TVM.[32]

4.2.9. Хранение, тик и такт транзакций. Транзакции хранилища очень похожи на фазу автономного хранения обычной транзакции. Транзакции Tick и tock похожи на обычные транзакции без фаз кредитования и отскока, потому что нет входящего сообщения.

4.2.10. Сплит-транзакции. Разделенные транзакции фактически состоят из двух транзакций. Если учетную запись ξ необходимо разделить на две учетные записи ξ и ξ':

- Сначала *транзакция split prepare, похожая на транзакцию tock (но в shardchain вместо masterchain), выдается для счета ξ. Это должна быть последняя транзакция для ξ* в блоке shardchain. Выход этапа обработки транзакции split prepare состоит не только из нового состояния счета *ξ, но и из нового состояния счета ξ', представленного сообщением конструктора к ξ'* (см. 4.1.7).

- Затем добавляется *транзакция split install* для учетной записи *ξ' со ссылкой на соответствующую транзакцию split prepare. Транзакция split install должна быть единственной транзакцией для ранее несуществующей учетной записи ξ'* в блоке. Он эффективно задает состояние ξ', как определено транзакцией split prepare.

4.2.11. Объединение транзакций. Транзакции слияния также состоят из двух транзакций каждая. Если учетная запись ξ' должна быть объединена в учетную запись ξ:

[32] Эталонная реализация эмулятора TVM, работающего в урезанной версии TVM, может быть зафиксирована в мастерчейне для использования при возникновении разногласий между валидаторами по конкретному запуску TVM. Таким образом, могут быть обнаружены дефектные реализации TVM. Затем эталонная реализация служит авторитетной
источник по оперативной семантике TBM. (См. [4, B.2])

- Сначала *транзакция merge prepare* выдается *для ξ', которая преобразует все свое постоянное состояние и баланс в специальное сообщение конструктора с назначением ξ* (см. 4.1.7).

- Затем *транзакция merge install* для *ξ, относящаяся к соответствующей транзакции merge prepare, обрабатывает сообщение конструктора. Транзакция merge install похожа на транзакцию tick в том, что она должна быть первой транзакцией для ξ* в блоке, но она находится в блоке shardchain, а не в masterchain, и имеет специальное входящее сообщение.

4.2.12. Сериализация общей транзакции. Любая транзакция содержит поля, перечисленные в 4.2.3. Как следствие, во всех транзакциях есть некоторые общие компоненты:

```
transaction$_ account_addr:uint256 lt:uint64 outmsg_cnt:uint15
   orig_status:AccountStatus end_status:AccountStatus
   in_msg:(Maybe ^(Message Any))
   out_msgs:(HashmapE 15 ^(Message Any))
   total_fees:Grams state_update:^(MERKLE_UPDATE Account)
   description:^TransactionDescr = Transaction;

!merkle_update#02 {X:Type} old_hash:uint256 new_hash:uint256
    old:^X new:^X = MERKLE_UPDATE X;
```

Восклицательный знак в декларации TL-B merkle_update указывает на специальную обработку, необходимую для таких значений. В частности, они должны храниться в отдельной ячейке, которая должна быть помечена как *экзотическая* битом в ее заголовке.
(ср. [4, 3.1]).

Полное объяснение сериализации *TransactionDescr, которая описывает одну транзакцию в соответствии с ее видом, перечисленным в 4.2.4, можно найти в 4.3.*

4.2.13. Представление исходящих сообщений, генерируемых транзакцией. Исходящие сообщения, генерируемые транзакцией *t,* хранятся в словаре

out_msgs с 15-битными ключами, равными 0, 1,..., $n - 1$, где n = outmsg_cnt — количество сгенерированных исходящих сообщений. Сообщение m_{i+1} с индексом $0 \le i < n$ должно иметь $Lt(m_{i+1}) = Lt(t) + i + 1$, и
Lt(t) = Lt−(t) явно хранится в поле lt.

4.2.14. Условия согласованности сделок. Общая сериализация полей, присутствующих в *Транзакции, независимо от ее типа и описания, позволяет нам наложить несколько «внешних» условий согласованности на любую транзакцию. Наиболее важный из них включает в себя поток значений* внутри транзакции: сумма всех входных данных (значение импорта входящего сообщения плюс исходный баланс счета) должна равняться сумме всех выходов (итоговый баланс счета, плюс сумма экспортных значений всех исходящих сообщений, плюс все хранилище, сборы за обработку и пересылку, взимаемые валидаторами). Таким образом, поверхностная проверка транзакции, которая обрабатывает входящее сообщение со значением импорта 1 грамм, полученное учетной записью с начальным балансом 10 граммов, генерируя исходящее сообщение со стоимостью экспорта 100 граммов в процессе, выявит его недействительность еще до проверки всех деталей выполнения TVM.

Другие условия согласованности могут незначительно зависеть от описания транзакции. Например, входящее сообщение, обрабатываемое обычной транзакцией, должно быть зарегистрировано в *InMsgDescr* охватывающего блока, а соответствующая запись должна содержать ссылку на эту транзакцию. Аналогичным образом, все исходящие сообщения, генерируемые всеми транзакциями (за исключением одного специального сообщения, сгенерированного разделенной транзакцией подготовки или подготовки слияния), должны быть зарегистрированы в *OutMsgDescr.*

4.2.15. Сбор всех операций по счету. Все транзакции в блоке, принадлежащем одному и тому же счету ξ, собираются в «блок связей счетов» *AccountBlock,* который по сути представляет собой словарные транзакции с 64-битными ключами, каждая из которых равна логическому времени соответствующей транзакции:

acc_trans$_ account_addr:uint256

transactions:(HashmapAug 64 ^Transaction Grams)
state_update:^(MERKLE_UPDATE Account)
= AccountBlock;

Словарь транзакций дополняется значением *граммов,* которое агрегирует общие сборы, собранные с этих транзакций.

В дополнение к этому словарю AccountBlock содержит обновление Merkle (см. [4, 3.1]) общего состояния учетной записи. Если аккаунт до блокировки не существовал, его состояние представлено account_none.

4.2.16. Условия согласованности для *AccountBlocks. Существует несколько общих условий согласованности, налагаемых на AccountBlock. В частности:*

- Транзакция, отображаемая как значение в расширенном словаре транзакций, должна иметь значение lt, равное ее ключу.
- Все транзакции должны принадлежать учетной записи, адрес account_addr которой указан в *AccountBlock.*
- Если t и t' являются двумя транзакциями с Lt(t) < Lt(t'), и их ключи последовательны в транзакциях, что означает, что нет транзакции t'' с Lt(t) < Lt(t'') < Lt(t'), то конечное состояние t должно соответствовать начальному состоянию t' (их хэши, явно указанные в обновлениях Меркла, должны быть равны).
- Если t является транзакцией с минимальным Lt(t), ее начальное состояние должно совпадать с начальным состоянием, как указано в state_update *AccountBlock.*
- Если t является транзакцией с максимальным Lt(t), ее конечное состояние должно совпадать с конечным состоянием, указанным в state_update *AccountBlock.*
- Список транзакций должен быть непустым.

Эти условия просто выражают тот факт, что состояние счета может измениться только в результате выполнения транзакции.

4.2.17. Сбор всех транзакций в блоке. Все транзакции в блоке представлены (см. 1.2.1):

_ (HashmapAugE 256 AccountBlock Grams) = ShardAccountBlocks;

4.2.18. Условия согласованности сбора всех транзакций. Опять же, на эту структуру накладываются условия согласованности, требующие, чтобы значение в ключе ξ было *AccountBlock* с адресом, равным ξ. *Дальнейшие условия согласованности связывают эту структуру с начальным и конечным состояниями шардчейна, указанными в блоке, требуя, чтобы:*

- Если *ShardAccountBlock* не имеет ключа ξ, *то состояние счета* ξ в начальном и конечном состоянии блока должно совпадать (или оно должно отсутствовать у обоих).
- Если ξ присутствует в *ShardAccountBlock, его начальное и конечное состояния, указанные в AccountBlock,* должны совпадать с состояниями, указанными в начальном и конечном состояниях блока shardchain, выраженных экземплярами *ShardAccounts* (см. 4.1.9).

Эти условия выражают, что состояние шардчейна действительно состоит из состояний отдельных счетов.

4.3 Описание транзакций

В этом разделе представлены конкретные схемы TL-B для описания транзакций в соответствии с классификацией, приведенной в разделе 4.2.4.

4.3.1. Причины исключения данных из описания транзакции. Описание транзакции для блокчейна с полной виртуальной машиной Тьюринга для выполнения смарт-контрактов обязательно является неполным. Действительно, действительно полное описание будет содержать все промежуточные состояния виртуальной машины после выполнения каждой инструкции, что не может поместиться в блок блокчейна разумного размера. Поэтому описание такой транзакции, скорее всего, будет содержать только общее количество шагов и хэши начального и конечного состояний виртуальной машины. Проверка такой транзакции обязательно будет включать в себя выполнение смарт-контракта для воспроизведения всех промежуточных шагов и конечного результата.

Если сжать последовательность всех промежуточных шагов виртуальной машины в только хэши начального и конечного состояний, то никаких деталей транзакции включать вообще не нужно: проверяющий, способный самостоятельно проверять выполнение виртуальной машины, также сможет проверить все остальные действия транзакции, начиная с ее исходных данных без этих деталей.

4.3.2. Причины включения данных в описание транзакции. Несмотря на вышеизложенные соображения, есть еще несколько причин для включения некоторых деталей в описание транзакции:

- Мы хотим наложить на транзакцию внешние условия согласованности, чтобы по крайней мере валидность потока значений внутри транзакции и валидность входящих и исходящих сообщений можно было быстро проверить без вызова виртуальной машины (см. 4.2.14). Это как минимум гарантирует инвариантность общего количества каждой криптовалюты в блокчейне, даже если не гарантирует правильность ее распределения.

- Мы хотим иметь возможность отслеживать основные изменения состояния учетной записи (например, ее создание, активацию или замораживание), проверяя данные, хранящиеся в описании

транзакции, не выясняя недостающие детали транзакции. Это упрощает проверку условий согласованности между состояниями accountchain и shardchain в блоке.

- Наконец, определенная информация, такая как общие шаги виртуальной машины, хэши ее начального и конечного состояний, общее потребление газа и код выхода, может значительно упростить отладку и реализацию программного обеспечения TON Blockchain. (Эта информация поможет программисту-человеку понять, что произошло.
  в определенном блоке блокчейна.)

С другой стороны, мы хотим минимизировать размер каждой транзакции, потому что мы хотим максимизировать количество транзакций, которые могут поместиться в каждый (ограниченный размер) блок. Поэтому вся информация, не требующаяся по одной из вышеуказанных причин, опускается.

4.3.3. Описание этапа хранения. Фаза хранения присутствует в нескольких видах транзакций, поэтому для этой фазы используется общее представление:

```
tr_phase_storage$_ storage_fees_collected:Grams
    storage_fees_due:(Maybe Grams)
    status_change:AccStatusChange
    = TrStoragePhase;
acst_unchanged$0 = AccStatusChange;   // x -> x
acst_frozen$10 = AccStatusChange;       // init -> frozen
acst_deleted$11 = AccStatusChange;      // frozen -> deleted
```

4.3.4. Описание кредитного этапа. Кредитный этап может привести к сбору некоторых причитающихся платежей:

```
tr_phase_credit$_ due_fees_collected:(Maybe Garms)
  credit:CurrencyCollection= TrCreditPhase;
```

Сумма due_fees_collected и кредита должна равна стоимости полученного сообщения плюс его ihr_fee, если сообщение не было получено с помощью ММСП (в противном случае ihr_fee присуждается валидаторам).

4.3.5. Описание этапа вычисления. Вычислительная фаза заключается в вызове TVM с правильными входами. В некоторых случаях TVM вообще не может быть вызван (например, если учетная запись отсутствует, не инициализирована или заморожена, а обрабатываемое входящее сообщение не имеет кода или полей данных или эти поля имеют неправильный хэш); это отражается соответствующими конструкторами.

```
tr_phase_compute_skipped$0 reason:ComputeSkipReason
   = TrComputePhase;
tr_phase_compute_vm$1 success:Bool msg_state_used:Bool
   account_activated:Bool gas_fees:Grams
   _:^[ gas_used:(VarUInteger 7)
   gas_limit:(VarUInteger 7) gas_credit:(Maybe (VarUInteger 3))
   mode:int8 exit_code:int32 exit_arg:(Maybe int32)
   vm_steps:uint32
   vm_init_state_hash:uint256 vm_final_state_hash:uint256 ]
   = TrComputePhase;
cskip_no_state$00 = ComputeSkipReason;
cskip_bad_state$01 = ComputeSkipReason;
cskip_no_gas$10 = ComputeSkipReason;
```

Конструкция TL-B _:^[...] описывает ссылку на ячейку, содержащую поля, перечисленные в квадратных скобках. Таким образом, несколько полей могут быть перемещены из ячейки, содержащей большую запись, в отдельную подячейку.

4.3.6. Пропущенная фаза вычислений. Если этап вычисления был пропущен, возможны следующие причины:

- Отсутствие средств на покупку газа.

- Отсутствие состояния (т.е. кода смарт-контракта и данных) как в учетной записи (несуществующей, неинициализированной или замороженной), так и в сообщении.

- Недопустимое состояние, переданное в сообщении (т. е. хэш состояния отличается от ожидаемого значения) в замороженную или неинициализированную учетную запись.

4.3.7. Допустимый этап вычисления. Если нет причин пропускать этап вычисления, вызывается TVM и результаты вычислений записываются в журнал. Возможны следующие параметры:

- Флаг успеха устанавливается тогда и только тогда, когда exit_code равно 0 или 1.

- Параметр msg_state_used отражает, использовалось ли состояние, переданное в сообщении. Если он установлен, флаг account_activated отражает, привело ли это к активации ранее замороженной, неинициализированной или несуществующей учетной записи.

- Параметр gas_fees отражает общую плату за газ, взимаемую валидаторами за выполнение этой транзакции. Он должен быть равен произведению gas_used и gas_price от текущего заголовка блока.

- Параметр gas_limit отражает предел газа для данного экземпляра TVM. Он равен меньшему из граммов, зачисленных на кредитном этапе, от стоимости входящего сообщения, деленного на текущую цену газа, либо к глобальному лимиту газа на транзакцию.

- Параметр gas_credit может быть ненулевым только для внешних входящих сообщений. Это меньшее количество газа, которое может быть оплачено с баланса счета, либо максимальный газовый кредит.

- Параметры exit_code и exit_args представляют значения состояния, возвращаемые TVM.

- Параметры vm_init_state_hash и vm_final_state_hash представляют собой хэши представления исходного и результирующего состояний

TVM, a vm_steps — общее количество шагов, выполняемых TVM (обычно равное двум плюс количество выполняемых инструкций, включая неявные RET).[33]

4.3.8. Описание фазы действия. Фаза действия происходит после допустимого этапа вычисления. Он пытается выполнить действия, сохраненные TVM на этапе вычисления, в *список действий. Он может завершиться ошибкой, поскольку список действий может оказаться слишком длинным, содержать недопустимые действия или действия, которые не могут быть выполнены (например, из-за нехватки средств для создания исходящего сообщения с требуемым значением).*

```
tr_phase_action$_ success:Bool valid:Bool no_funds:Bool
  status_change:AccStatusChange
  total_fwd_fees:(Maybe Grams) total_action_fees:(Maybe Grams)
  result_code:int32 result_arg:(Maybe int32) tot_actions:int16
  spec_actions:int16 msgs_created:int16
  action_list_hash:uint256 tot_msg_size:StorageUsed
  = TrActionPhase;
```

4.3.9. Описание фазы отскока.

```
tr_phase_bounce_negfunds$00 = TrBouncePhase;
tr_phase_bounce_nofunds$01 msg_size:StorageUsed
    req_fwd_fees:Grams= TrBouncePhase;
tr_phase_bounce_ok$1 msg_size:StorageUsed
     msg_fees:Grams fwd_fees:Grams= TrBouncePhase;
```

4.3.10. Описание обычной сделки.

```
trans_ord$0000 storage_ph:(Maybe TrStoragePhase)
  credit_ph:(Maybe TrCreditPhase)
```

[33] Обратите внимание, что эта запись не представляет собой изменение состояния учетной записи, поскольку транзакция все еще может быть прервана на этапе действия. В этом случае новые постоянные данные, на которые косвенно ссылается vm_final_state_hash, будут отброшены.

```
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool bounce:(Maybe TrBouncePhase)
  destroyed:Bool
  = TransactionDescr;
```

На эту структуру накладывается несколько условий согласованности:

- действие отсутствует тогда и только тогда, когда этап вычисления был неудачным.
- Прерванный флаг устанавливается либо в том случае, если фаза действия отсутствует, либо если фаза действия была неудачной.
- Фаза возврата происходит только в том случае, если установлен флаг прерванного действия и входящее сообщение было отклонено.

4.3.11. Описание транзакции хранения. Транзакция хранилища состоит только из одного этапа автономного хранилища:

```
trans_storage$0001 storage_ph:TrStoragePhase
  = TransactionDescr;
```

4.3.12. Описание тиковых и так-транзакций. Транзакции Tick и tock похожи на обычные транзакции без входящего сообщения, поэтому нет фаз кредитования или отказа:

```
trans_tick_tock$001 is_tock:Bool storage:TrStoragePhase
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool = TransactionDescr;
```

4.3.13. Сплит подготавливает и устанавливает транзакции. Транзакция split prepare аналогична транзакции tock в мастерчейне, но она должна генерировать ровно одно специальное сообщение конструктора; в противном случае фаза действия прерывается.

```
split_merge_info$_ cur_shard_pfx_len:(## 6)
```

```
  acc_split_depth:(## 6) this_addr:uint256 sibling_addr:uint256
  = SplitMergeInfo;
trans_split_prepare$0100 split_info:SplitMergeInfo
  compute_ph:TrComputePhase action:(Maybe^ TrActionPhase)
  aborted:Bool destroyed:Bool
  = TransactionDescr;
trans_split_install$0101 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  installed:Bool = TransactionDescr;
```

Обратите внимание, что транзакция split install для новой учетной записи ξ' относится к соответствующей транзакции split prepare ранее существующей учетной записи ξ.

4.3.14. Слияние подготавливает и устанавливает транзакции. Транзакция подготовки слияния преобразует состояние и баланс учетной записи в сообщение, а последующая транзакция установки слияния обрабатывает следующее состояние:

```
trans_merge_prepare$0110 split_info:SplitMergeInfo
  storage_ph:TrStoragePhase aborted:Bool
  = TransactionDescr;
trans_merge_install$0111 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe^TrActionPhase)
  aborted:Bool destroyed:Bool
  = Transaction Descr;
```

4.4 Вызов смарт-контрактов в TVM

В этом разделе описываются точные параметры, с которыми TVM вызывается на этапе вычислений обычных и других транзакций.

4.4.1. Код смарт-контракта. *Код* смарт-контракта обычно является частью постоянного состояния учетной записи, по крайней мере, если учетная запись активна (см. 4.1.6). Однако замороженный или неинициализированный (или несуществующий) счет не имеет постоянного состояния, за исключением, возможно, баланса счета и хэша его предполагаемого состояния (равного адресу счета для неинициализированных счетов). В этом случае код должен быть указан в поле init входящего сообщения, обрабатываемого транзакцией (см. 3.1.7).

4.4.2. Постоянные данные смарт-контракта. Постоянные *данные* смарт-контракта хранятся вместе с его кодом, и применяются замечания, аналогичные тем, которые сделаны выше в 4.4.1. В этом отношении код и постоянные данные смарт-контракта являются лишь двумя частями его постоянного состояния, которые отличаются только тем, как они обрабатываются TVM во время выполнения смарт-контракта.

4.4.3. Среда библиотеки смарт-контрактов. Библиотечная *среда* смарт-контракта представляет собой хэш-карту, отображающую 256-битные хэши ячеек (представления) в сами соответствующие ячейки. При доступе к внешней ссылке ячейки во время выполнения смарт-контракта ячейка, на которую ссылается ячейка, просматривается в среде библиотеки, и внешняя ссылка ячейки прозрачно заменяется найденной ячейкой.

Библиотечная среда для вызова смарт-контракта вычисляется следующим образом:

A. Глобальная библиотечная среда для рассматриваемой рабочей цепи взята из текущего состояния мастерчейна.[34]

[34] Наиболее распространенным способом создания разделяемых библиотек для TVM является публикация ссылки на корневую ячейку библиотеки в мастерчейне.

Б. Далее он дополняется локальной библиотечной средой смарт-контракта, хранящейся в библиотечном поле состояния смарт-контракта. Учитываются только 256-битные ключи, равные хэшам соответствующих значений ячеек. Если ключ присутствует как в глобальной, так и в локальной библиотечной средах, локальная среда имеет приоритет при объединении двух библиотечных сред.

В. Наконец, библиотека сообщений, хранящаяся в поле библиотеки поля init входящего сообщения, также учитывается. Обратите внимание, однако, что если учетная запись заморожена или не инициализирована, поле библиотеки сообщения является частью предлагаемого состояния учетной записи и используется вместо локальной среды библиотеки на предыдущем шаге. Библиотека сообщений имеет более низкий приоритет, чем локальная и глобальная библиотечные среды.

4.4.4. Исходное состояние TBM. Новый экземпляр TVM инициализируется перед выполнением смарт-контракта следующим образом:

- Исходный cc (текущее продолжение) инициализируется с помощью среза ячейки, созданного из кода ячейки, содержащего код смарт-контракта, вычисляемого, как описано в 4.4.1.

- Значение cp (кодовая страница TVM) равно нулю. Если смарт-контракт хочет использовать другую кодовую страницу TVM *x, он должен переключиться на нее, используя SETCODEPAGE x* в качестве первой инструкции своего кода.

- Регистр управления c0 (возвращаемое продолжение) инициализируется экстраординарным продолжением ec_quit с параметром 0. При выполнении это продолжение приводит к окончанию TVM с кодом выхода 0.

- Регистр управления c1 (альтернативное возвращаемое продолжение) инициализируется экстраординарным продолжением ec_quit с параметром 1. При вызове это приводит к завершению TVM с кодом

выхода 1. (Обратите внимание, что завершение с кодом выхода 0 или 1 считается успешным прекращением.)

- Регистр управления c2 (обработчик исключений) инициализируется экстраординарным продолжением ec_quit_exc. При вызове он берет верхнее целое число из стека (равное номеру исключения) и завершает TVM кодом выхода, равным этому целому числу. Таким образом, по умолчанию все исключения прекращают выполнение смарт-контракта с кодом выхода, равным номеру исключения.

- Регистр управления c3 (словарь кода) инициализируется ячейкой с кодом смарт-контракта, аналогично начальному текущему продолжению (cc).

- Регистр управления c4 (корень постоянных данных) инициализируется постоянными данными смарт-контракта.[35]

- Регистр управления c5 (корень действий) инициализируется пустой ячейкой. Примитивы «выходного действия» TVM, такие как SENDMSG, используют c5 для накопления списка действий (например, исходящих сообщений), которые должны быть выполнены после успешного прекращения смарт-контракта (см. 4.2.7 и 4.2.8).

- Регистр управления c7 (корень временных данных) инициализируется одноэлементным *кортежем, единственным компонентом которого является Кортеж,* содержащий экземпляр *SmartContractInfo* со смарт-контрактным балансом и другой полезной информацией (см. 4.4.10). Смарт-контракт может заменить временные данные, особенно все компоненты *Кортежа* в c7, кроме первого, любыми другими временными данными, которые ему могут потребоваться. Однако исходное содержимое *SmartContractInfo* в первом компоненте *Кортежа,* хранящемся в c7, проверяется и иногда

[35] Постоянные данные смарт-контракта не должны загружаться полностью, чтобы это произошло. Вместо этого загружается корень, и TVM может загружать другие ячейки своими ссылками из корня только при доступе к ним, обеспечивая тем самым форму виртуальной памяти.

модифицируется примитивами SENDMSG TVM и другими примитивами «выходного действия» TVM.

- Газовые пределы *gas* = (g_m,g_l,g_c,g_r) инициализируются следующим образом:
  - Максимальный *лимит газа* gm устанавливается на меньший из общего баланса Gram смарт-контракта (после кредитной фазы, то есть в сочетании со значением входящего сообщения), деленный на текущую цену газа, либо глобальный лимит газа на исполнение.[36]
  - Текущий *предел газа* g_l устанавливается на меньшее значение либо Грамма входящего сообщения, деленное на цену газа, либо глобального предела газа на исполнение. Таким образом, всегда $g_l \leq g_m$. *Для входящих внешних сообщений* $g_l = 0$, так как они не могут нести никакого значения.
  - Газовый *кредит* g_c устанавливается равным нулю для входящих внутренних сообщений и меньшим из gm или фиксированным малым значением (по умолчанию внешнее сообщение gas credit, настраиваемый параметр) для входящих внешних сообщений.
  - Наконец, оставшийся *предел газа* g_r автоматически инициализируется $g_l + g_c$.

4.4.5. Исходный стек TVM для обработки внутреннего сообщения. После инициализации TVM, как описано в 4.4.4, его стек инициализируется путем отправки аргументов в функцию main() смарт-контракта следующим образом:

- Граммовый баланс b смарт-контракта (после зачисления значения входящего сообщения) передается как *целое* количество нанограммов.

[36] Как глобальный лимит газа, так и цена на газ являются настраиваемыми параметрами, определяемыми текущим состоянием мастерчейна.

- Грамм баланса b_m входящего сообщения m передается как *целое количество* нанограммов.
- Входящее сообщение m передается в виде ячейки, содержащей сериализованное значение типа *Message X, где X* — тип тела сообщения.
- Тело m_b : X входящего сообщения, равное значению тела поля m, *передается как срез ячейки.*
- Наконец, *селектор функций s, целое число*, обычно равное нулю, помещается в стек.

После этого выполняется код смарт-контракта, равный его начальному значению c3. Он выбирает правильную функцию в соответствии с *s, которая, как ожидается, обработает оставшиеся аргументы функции и завершится впоследствии.*

4.4.6. Обработка входящего внешнего сообщения. Входящее внешнее сообщение обрабатывается аналогично разделам 4.4.4 и 4.4.5 со следующими изменениями:

- Селектору функций s присвоено значение -1, а не 0.
- Баланс Gram b_m входящего сообщения всегда равен 0.
- Начальный предел тока газа g_l всегда равен 0. Однако первоначальный газовый кредит $g_c > 0$.

Смарт-контракт должен расторгнуться с $g_c = 0$ или $g_r \geq g_c$*; в противном случае транзакция и блок, содержащий ее, являются недействительными. Валидаторы или коллаторы, предполагающие кандидат на блок, никогда не должны включать транзакции, обрабатывающие входящие внешние сообщения, которые являются недопустимыми.*

4.4.7. Обработка тиковых и такковых транзакций. Стек TVM для обработки тиковых и так-транзакций (см. 4.2.4) инициализируется нажатием следующих значений:

- Граммовый баланс b текущего счета в нанограммах (*целое число).*
- 256-битный адрес ξ текущего счета внутри мастерчейна, представленный целым числом без знака.
- Целое число, равное 0 для тиковых транзакций и -1 для транзакций tock.
- Селектор функций *s, равный* -2.

4.4.8. Обработка сплит-подготовки транзакций. Для обработки транзакций split prepare (см. 4.3.13) стек TVM инициализируется нажатием следующих значений:

- Граммовый остаток b текущего счета.
- Фрагмент *,* содержащий *SplitMergeInfo* (см. 4.3.13).
- 256-битный адрес ξ текущего счета.
- 256-разрядный адрес $\tilde{\xi}$ учетной записи брата или сестры.
- Целое число $0 \le d \le 63$, равное положению единственного бита, в котором ξ и $\tilde{\xi}$ отличаются.
- Селектор функций *s, равный* -3.

4.4.9. Обработка слияний установочных транзакций. Для обработки транзакций merge install (см. 4.3.14) стек TVM инициализируется путем отправки следующих значений:

- Граммовый остаток b текущего счета (уже в сочетании с Граммовым балансом счета брата или сестры).
- Граммовый баланс b' счета братьев и сестер, взятый из входящего сообщения m*.*
- Сообщение m из родительской учетной записи, автоматически генерируемое транзакцией подготовки слияния. Поле init содержит конечное состояние $\tilde{S}$ родительского счета.

- Положение $S\tilde{}$ родственной учетной записи, представленное *StateInit* (см. 3.1.7).
- Фрагмент , содержащий *SplitMergeInfo* (см. 4.3.13).
- 256-битный адрес ξ текущего счета.
- 256-разрядный адрес $\xi\tilde{}$ учетной записи брата или сестры.
- Целое число $0 \le d \le 63$, равное положению единственного бита, в котором ξ и $\xi\tilde{}$ отличаются.
- Селектор функций *s, равный* -4.

4.4.10. Информация о смарт-контракте. Информационная структура смарт-контракта *SmartContractInfo, переданная в первом компоненте Кортежа,* содержащемся в управляющем регистре c7, также является *Кортежем*, содержащим следующие данные:

```
[ magic:0x076ef1ea actions:Integer msgs_sent:Integer
  unixtime:Integer block_lt:Integer trans_lt:Integer
  rand_seed:Integer balance_remaining:[Integer (Maybe Cell)]
  myself:MsgAddressInt global_config:(Maybe Cell)
] = SmartContractInfo;
```

Другими словами, первый компонент этого кортежа — это магия *целочисленного числа*, всегда равная 0x076ef1ea, второй компонент — *целочисленные* действия, первоначально инициализированные нулем, но приращенные на единицу всякий раз, когда выходное действие устанавливается примитивом выходного действия TVM, отличным от RAW, и так далее. Оставшийся баланс представлен парой, т.е. двухкомпонентным *кортежем: первый компонент представляет собой наноГраммовый баланс, а второй компонент представляет собой словарь с 32-битными ключами, представляющими все остальные валюты, если таковые имеются (см. 3.1.6).*

Поле rand_seed (256-битное целое число без знака) здесь инициализируется детерминированно, начиная с rand_seed блока, адреса учетной записи, хэша обрабатываемого входящего сообщения (если таковое имеется) и логического времени транзакции trans_lt.

4.4.11. Сериализация выходных действий. Выходные *действия* смарт-контракта аккумулируются в связанном списке, хранящемся в управляющем регистре c5. Список выходных действий сериализуется как значение типа *OutList n, где n* — длина списка:

```
out_list_empty$_ = OutList 0;
out_list$_ {n:#} prev:^(OutList n) action:OutAction
   = OutList (n + 1);
action_send_msg#0ec3c86d out_msg:^(Message Any) = OutAction;
action_set_code#ad4de08e new_code:^Cell = OutAction;
```

5 Компоновка блоков

В этой главе представлен макет блока, используемый блокчейном TON, объединяющий структуры данных, описанные отдельно в предыдущих главах, чтобы получить полное описание блока shardchain. В дополнение к схемам TL-B, определяющим представление блока shardchain деревом ячеек, в этой главе описываются точные форматы сериализации для результирующих пакетов (коллекций) ячеек, которые необходимы для представления блока shardchain в виде файла.

Блоки Masterchain похожи на блоки shardchain, но имеют некоторые дополнительные поля. Необходимые изменения рассматриваются отдельно в разделе 5.2.

5.1 Схема блока Шардчейн

В этом разделе перечислены структуры данных, которые должны содержаться в блоке shardchain и в состоянии shardchain, и завершается представлением формальной схемы TL-B для блока shardchain.

5.1.1. Компоненты состояния шардчейна. Штат шардчейн состоит из:

- *ShardAccounts, разделенная часть состояния shardchain (см. 1.2.2), содержащая состояние всех учетных записей, назначенных этому сегменту (см. 4.1.9).*
- *OutMsgQueue, очередь выходных сообщений shardchain (см. 3.3.6).*
- *SharedLibraries, описание всех разделяемых библиотек shardchain (пока непустые только в мастерчейне).*
- Логическое время и unixtime последнего изменения состояния.
- Общий баланс сегмента.
- Хэш-ссылка на самый последний блок мастерчейна, косвенно описывающая состояние мастерчейна и, через него, состояние всех других шардчейнов блокчейна TON (ср. 1.5.2).

5.1.2. Компоненты шардчейного блока. Блок shardchain должен содержать:

- Список *проверяющих подписей* (см. 1.2.6), который является внешним по отношению ко всему остальному содержимому блока.
- *BlockHeader, содержащий общую информацию о блоке (см. 1.2.5)*
- Хэш ссылается на непосредственно предшествующий блок или блоки того же шардчейна, а также на самый последний блок мастерчейна.
- *InMsgDescr* и *OutMsgDescr, дескрипторы входящих и исходящих сообщений (см. 3.2.8 и 3.3.5).*
- *ShardAccountBlocks, сбор всех транзакций, обработанных в блоке (см. 4.2.17) вместе со всеми обновлениями состояний учетных записей, назначенных сегменту. Это разделенная* часть блока шардчейна (ср. 1.2.2).
- Поток значений, *описывающий общее значение, импортированное из предыдущих блоков одной и той же цепочки сегментов и из входящих сообщений, общее значение, экспортируемое исходящим сообщением, общее значение, собранное проверяющими, и общее значение, оставшееся в сегменте.*
- Обновление *Меркла* (ср. [4, 3.1]) состояния шардчейна. Такое обновление Меркла содержит хэши начального и конечного состояний шардчейна относительно блока, а также все новые ячейки конечного состояния, которые были созданы при обработке блока.[37]

5.1.3. Общие части блочной компоновки для всех рабочих цепей. Напомним, что разные рабочие цепи могут определять собственные правила обработки сообщений, другие типы транзакций, другие

[37] В принципе, экспериментальная версия TON Blockchain может предпочесть сохранить только хэши начального и конечного состояний шардчейна. Обновление Merkle увеличивает размер блока, но оно удобно для полных узлов, которые хотят сохранить и обновить свою копию состояния shardchain. В противном случае полным узлам пришлось бы повторять все вычисления, содержащиеся в блоке, чтобы самостоятельно вычислить обновленное состояние shardchain.

компоненты состояния и другие способы сериализации всех этих данных. Однако некоторые компоненты блока и его состояние должны быть общими для всех рабочих цепей, чтобы поддерживать взаимодействие между различными рабочими цепочками. К таким общим компонентам относятся:

- *OutMsgQueue, очередь исходящих сообщений shardchain, которая сканируется соседними shardchains на наличие адресованных им сообщений.*
- Внешняя структура *InMsgDescr* в виде хэш-карты с 256-битными ключами, равными хэшам импортированных сообщений. (Сами дескрипторы входящих сообщений не обязательно должны иметь одинаковую структуру.)
- Некоторые поля в заголовке блока, идентифицирующие shardchain и блок, а также пути от заголовка блока к другой информации, указанной в этом списке.
- Сведения о потоке значений.

5.1.4. Схема TL-B для состояния шардчейн. Состояние шардчейна (см. 1.2.1 и 5.1.1) сериализуется по следующей схеме TL-B:

```
ext_blk_ref$_ start_lt:uint64 end_lt:uint64
   seq_no:uint32 хэш:uint256 = ExtBlkRef;

master_info$_ master:ExtBlkRef = BlkMasterInfo;

shard_ident$00 shard_pfx_bits:(## 6)
   workchain_id:int32 shard_prefix:uint64 = ShardIdent;

shard_state shard_id:ShardIdent
   out_msg_queue:OutMsgQueue
   total_balance:CurrencyCollection
   total_validator_fees:CurrencyCollection
   accounts:ShardAccounts
```

```
  libraries:(HashmapE 256 LibDescr)
  master_ref:(Maybe BlkMasterInfo)
  custom:(Maybe ^McStateExtra)
  = ShardState;
```

Пользовательское поле обычно присутствует только в мастерчейне и содержит все данные, специфичные для мастерчейна. Однако другие рабочие цепи могут использовать ту же ссылку на ячейку для ссылки на свои конкретные данные о состоянии.

5.1.5. Описание разделяемых библиотек. Разделяемые библиотеки в настоящее время могут присутствовать только в блоках мастерчейна. Они описываются экземпляром *HashmapE(256,LibDescr), где 256-битный ключ является хэшем представления библиотеки, а LibDescr* описывает одну библиотеку:

```
shared_lib_descr$00 lib:^Cell publishers:(Hashmap 256 True)
  = LibDescr;
```

Здесь publishers представляет собой хэш-карту с ключами, равными адресам всех учетных записей, опубликовавших соответствующую разделяемую библиотеку. Разделяемая библиотека сохраняется до тех пор, пока хотя бы одна учетная запись хранит ее в своей коллекции опубликованных библиотек.

5.1.6. Схема TL-B для блока неподписанного шардчейна. Точный формат *беззнакового* (ср. 1.2.6) блока шардчейна задается следующей схемой TL-B:

```
block_info версия:uint32
  not_master:(## 1)
  after_merge:(## 1) before_split:(## 1) flags:(## 13)
  seq_no:# vert_seq_no:#
  shard:ShardIdent gen_utime:uint32
  start_lt:uint64 end_lt:uint64
  master_ref:not_master?^BlkMasterInfo
  prev_ref:seq_no?^(BlkPrevInfo after_merge)
  prev_vert_ref:vert_seq_no?^(BlkPrevInfo 0)
  = BlockInfo;
```

```
prev_blk_info#_ {merged:#} prev:ExtBlkRef
   prev_alt:merged? ExtBlkRef = BlkPrevInfo merged;

unsigned_block info:^BlockInfo value_flow:^ValueFlow
   state_update:^(MERKLE_UPDATE ShardState)
   extra:^BlockExtra = Block;

block_extra in_msg_descr:^InMsgDescr
    out_msg_descr:^OutMsgDescr
    account_blocks:ShardAccountBlocks
    rand_seed:uint256
    custom:(Maybe ^McBlockExtra) = BlockExtra;
```

Пользовательское поле обычно присутствует только в мастерчейне и содержит все данные, специфичные для мастерчейна. Однако другие рабочие цепи могут использовать ту же ссылку на ячейку для ссылки на свои конкретные блочные данные.

5.1.7. Описание общего значения потока через блок. Общий поток значений через блок сериализуется по следующей схеме TL-B:

```
value_flow _:^[ from_prev_blk:CurrencyCollection
    to_next_blk: CurrencyCollection
    imported: CurrencyCollection exported: CurrencyCollection ]
    fees_collected: CurrencyCollection
    _:^[
    fees_imported: CurrencyCollection
    created: CurrencyCollection
    minted: CurrencyCollection
    ] = ValueFlow;
```

Напомним, что _:^[...] — это конструкция TL-B, указывающая на то, что группа полей была перемещена в отдельную ячейку. Последние три поля могут быть ненулевыми только в блоках мастерчейна.

5.1.8. Подписанный шардчейн блок. Подписанный блок shardchain — это просто неподписанный блок, дополненный коллекцией подписей валидатора:

```
ed25519_signature#5 R:uint256 s:uint256 = CryptoSignature;

signed_block block:^Block blk_serialize_hash:uint256
   signatures:(HashmapE 64 CryptoSignature)
   = SignedBlock;
```

Хэш сериализации blk_serialize_hash неподписанного блока *block* по существу является хэшем конкретной сериализации блока в строку октета (см. 5.3.12 для более подробного объяснения). Подписи, собранные в подписях, представляют собой подписи Ed25519 (ср. A.3), сделанные с помощью закрытых ключей валидатора sha256 объединения 256-битного хэша представления блока и его 256-битного хэш-blk_serialize_hash сериализации. 64-битные ключи в словарных сигнатурах представляют собой первые 64 бита открытых ключей соответствующих валидаторов.

5.1.9. Сериализация подписанного блока. Общая процедура сериализации и подписания блока может быть описана следующим образом:

А. Генерируется беззнаковый блок B, преобразуется в полный мешок ячеек (см. 5.3.2) и сериализуется в строку октета S_B.

Б. Валидаторы подписывают 256-битный комбинированный хэш

$$HB := \text{sha256}\,(\text{Hash}_\infty(B).\ \text{Hash}_M(S_B)) \qquad (18)$$

хэша представления B и хэша Меркла его сериализации S_B.

В. Подписанный шардчейн блок $B^{\sim}$ генерируется из B и этих сигнатур валидатора, как описано выше (см. 5.1.8).

Г. Этот знаковый блок $B^{\sim}$ преобразуется в неполный мешок ячеек, который содержит только сигнатуры валидатора, но сам неподписанный блок отсутствует в этом мешке ячеек, являясь его единственной отсутствующей ячейкой.

Д. Этот неполный мешок ячеек сериализуется, и его сериализация предшествует ранее построенной сериализации беззнакового блока.

Результатом является сериализация знакового блока в строку октета. Он может распространяться по сети или храниться в файле диска.

5.2 Макет блока Мастерчейн

Блоки Masterchain очень похожи на блоки shardchain основной рабочей цепи. В этом разделе перечислены некоторые изменения, необходимые для получения описания блока мастерчейна из описания блока shardchain, приведенного в 5.1.

5.2.1. Дополнительные компоненты, присутствующие в состоянии мастерчейна. В дополнение к компонентам, перечисленным в 5.1.1, состояние мастерчейна должно содержать:

- *ShardHashes* — описывает текущую конфигурацию сегментов и содержит хэши последних блоков соответствующих цепей сегментов.
- *ShardFees* — описывает общую сумму сборов, собираемых валидаторами каждого шардчейна.
- *ShardSplitMerge* — описывает будущие события разделения/слияния сегментов. Он сериализуется как часть *ShardHashes.*
- *ConfigParams* — Описывает значения всех настраиваемых параметров блокчейна TON.

5.2.2. Дополнительные компоненты, присутствующие в блоках мастерчейна. Помимо компонентов, перечисленных в 5.1.2, каждый блок мастерчейна должен содержать:

- *ShardHashes* — описывает текущую конфигурацию сегментов и содержит хэши последних блоков соответствующих цепей сегментов. (Обратите внимание, что этот компонент также присутствует в состоянии мастерчейна.)

5.2.3. Описание *ShardHashes. ShardHashes* представлен словарем с 32-битными *workchain_ids в качестве ключей и «двоичными деревьями сегментов», представленными TL-B типа BinTree ShardDescr, в качестве значений. Каждый лист этого двоичного дерева сегментов содержит значение типа ShardDescr, которое описывает один сегмент, указывая порядковый номер seq_no, логический литр времени и хэш-хэш последнего (подписанного) блока соответствующего шардчейна.*

```
bt_leaf$0 {X:Type} leaf:X = BinTree X;
bt_fork$1 {X:Type} left:^(BinTree X) right:^(BinTree X)
                = BinTree X;
fsm_none$0 = FutureSplitMerge;
fsm_split$10 mc_seqno:uint32 = FutureSplitMerge;
fsm_merge$11 mc_seqno:uint32 = FutureSplitMerge;

shard_descr$_ seq_no:uint32 lt:uint64 hash:uint256
   split_merge_at:FutureSplitMerge = ShardDescr;

_ (HashmapE 32 ^(BinTree ShardDescr)) = ShardHashes;
```

Поля mc_seqno fsm_split и fsm_merge используются для обозначения будущих событий слияния или разделения сегментов. Блоки Шардчейна, относящиеся к блокам мастерчейна с порядковыми номерами до, но не включая, указанные в mc_seqno генерируются обычным способом. После достижения указанного порядкового номера должно произойти событие слияния или разделения сегментов.

Обратите внимание, что сам мастерчейн опущен из *ShardHashes* (т.е. 32-битный индекс -1 отсутствует в этом словаре).

5.2.4. Описание *ShardFees. ShardFees* - это структура мастерчейна, используемая для отражения общих сборов, собранных до сих пор валидаторами шардчейна. Общие комиссии, отраженные в этой структуре, накапливаются в мастерчейне путем зачисления их на специальный счет, адрес которого является настраиваемым параметром. Обычно эта учетная запись представляет собой смарт-контракт, который вычисляет и распределяет вознаграждения среди всех валидаторов.

```
bta_leaf$0 {X:Type} {Y:Type} leaf:X extra:Y = BinTreeAug X Y;
bta_fork$1 {X:Type} {Y:Type} left:^(BinTreeAug X Y)
            right:^(BinTreeAug X Y) extra:Y = BinTreeAug X Y;
_ (HashmapAugE 32 ^(BinTreeAug True CurrencyCollection)
  CurrencyCollection) = ShardFees;
```

Структура ShardFees аналогична структуре *ShardHashes* (ср. 5.2.3), но словарные и двоичные деревья дополнены значениями валют, равными значениям total_validator_fees конечных состояний соответствующих блоков shardchain. Значение, агрегированное в корне *ShardFees*, добавляется вместе с total_validator_fees состояния мастерчейна, что дает общую плату за валидатор TON Blockchain. Увеличение значения, агрегированного в корне *ShardFees* от начального до конечного состояния блока мастерчейна, отражается в fees_imported в потоке значений этого блока мастерчейна.

5.2.5. Описание *ConfigParams. Напомним, что настраиваемые параметры* или *конфигурационный словарь* представляет собой конфигурацию словаря с 32-битными ключами, хранящимися внутри первой ячейки ссылки на постоянные данные конфигурационного смарт-контракта γ (см. 1.6). Адресная γ смарт-контракта конфигурации и копия конфигурационного словаря дублируются в полях config_addr и конфигурации *структуры ConfigParams,* явно включенных в состояние мастерчейна для облегчения доступа к текущим значениям настраиваемых параметров (см. 1.6.3):

```
_ config_addr:uint256 config:^(Hashmap 32 ^Cell)
```

= ConfigParams;

5.2.6. Данные о состоянии Мастерчейна. Данные, специфичные для состояния мастерчейна, собираются в *McStateExtra, уже упомянутом в 5.1.4:*

masterchain_state_extra#cc1f
shard_hashes:ShardHashes
shard_fees:ShardFees
config:ConfigParams
= McStateExtra;

5.2.7. Данные блока Masterchain. Аналогично, данные, специфичные для блоков мастерчейна, собираются в *McBlockExtra:*

masterchain_block_extra#cc9f
shard_hashes:ShardHashes
= McBlockExtra;

5.3 Сериализация мешка ячеек

Описание, приведенное в предыдущем разделе, определяет способ представления блока shardchain в виде дерева ячеек. Однако это дерево ячеек должно быть сериализовано в файл, подходящий для дискового хранения или передачи по сети. В этом разделе обсуждаются стандартные способы сериализации дерева, группы обеспечения доступности баз данных или пакета ячеек в строку октета.

5.3.1. Превращение дерева клеток в мешок клеток. Напомним, что значения произвольных (зависимых) алгебраических типов данных представлены в блокчейне TON деревьями *ячеек. Такое дерево клеток преобразуется в направленный ациклический граф, или DAG, клеток, путем идентификации идентичных клеток в дереве. После этого мы могли бы заменить каждую из ссылок каждой ячейки 32-байтным хэшем представления упомянутой ячейки и получить мешок ячеек. По соглашению, корень исходного дерева клеток является отмеченным элементом результирующего мешка клеток, так что любой, кто получает этот мешок клеток и знает помеченный элемент, может реконструировать исходный DAG клеток, следовательно, также оригинальное дерево клеток.*

5.3.2. Полные мешки ячеек. Предположим, что мешок клеток является *полным,* если он содержит все клетки, на которые ссылается любая из его клеток. Другими словами, полный мешок клеток не имеет каких-либо «нерешенных» хэш-ссылок на ячейки за пределами этого мешка клеток. В большинстве случаев нам нужно сериализовать только полные мешки клеток.

5.3.3. Внутренние ссылки внутри ячейки. Предположим, что ссылка на ячейку c, принадлежащую мешку клеток B, является *внутренней (по отношению к B),* если ячейка ci, на которую ссылается эта ссылка, также принадлежит B. В противном случае ссылка называется *внешней. Мешок клеток является полным тогда и только тогда, когда все ссылки на составляющие его клетки являются внутренними.*

5.3.4. Присвоение индексов клеткам из мешка клеток. Пусть $c_0,\ldots, c_{n-1}$ — n различных клеток, принадлежащих мешку клеток B. *Мы можем перечислить эти ячейки в некотором порядке,* а затем присвоить индексы от 0 до $n - 1$, так что ячейка c_i получает индекс i. *Некоторые варианты заказа ячеек:*

- Упорядочивайте ячейки по хэшу их представления. Затем $\text{Hash}(c_i) < \text{Hash}(c_j)$ всякий раз, когда $i < j$.
- Топологический порядок: если ячейка c_i относится к ячейке c_j, *то* $i < j$. *В общем, существует более одного топологического порядка для одного и того же мешка ячеек. Существует два стандартных способа построения топологических порядков:*
  - Порядок «глубина-первый»: примените поиск по глубине к направленному ациклическому графу ячеек, начиная с его корня (т. е. помеченной ячейки), и перечислите ячейки в том порядке, в котором они посещены.
  - Первый порядок ширины: такой же, как и выше, но с применением поиска по ширине.

Обратите внимание, что топологический порядок всегда присваивает индекс 0 корневой ячейке пакета клеток, построенного из дерева клеток. В большинстве случаев мы предпочитаем использовать топологический порядок или порядок глубины, если мы хотим быть более конкретными.

Если клетки перечислены в топологическом порядке, то проверка того, что в мешке клеток нет циклических ссылок, является немедленной. С другой стороны, упорядочивание ячеек по хэшу их представления упрощает проверку отсутствия дубликатов в сериализованном пакете ячеек.

5.3.5. Схема процесса сериализации. Процесс сериализации мешка клеток B, состоящего из n ячеек, можно очертить следующим образом:

1. Перечислите ячейки из B в топологическом порядке: $c_0, c_1,\ldots, c_{n-1}$. *Тогда* c_0 является корневой клеткой B.

2. Выберите целое число *s, такое, чтобы* $n \le 2^s$. Представьте каждую ячейку c_i интегральным числом октетов стандартным способом (ср. 1.1.3 или [4, 3.1.4]), но используя беззнаковое большое порядковое *s-битное целое число j вместо* хэш-Hash(c_j) для представления внутренних ссылок на ячейку c_j (см. 5.3.6 ниже).

3. Сцепляйте представления клеток c_i, полученные таким образом в порядке возрастания *i.*

4. Опционально может быть построен индекс, состоящий из $n + 1$ *t-бит целых записей* $L_0,\ldots, L_n$, *где* L_i — общая длина (в октетах) представлений ячеек c_j с $j \le i$, *а целое число* $t \ge 0$ выбрано так, что $L_n \le 2^t$.

5. Сериализация мешка ячеек теперь состоит из магического числа, обозначающего точный формат сериализации, за которым следуют целые числа $s \ge 0$, $t \ge 0$, $n \le 2^s$, необязательный индекс, состоящий из $\lceil (n+1)t/8 \rceil$ октетов, и L_n октетов с представлениями ячеек.

6. Дополнительный CRC32 может быть добавлен к сериализации в целях проверки целостности.

Если индекс включен, любая ячейка c_i в сериализованном мешке ячеек может быть легко доступна по ее индексу *i* без десериализации всех других ячеек или даже без загрузки всего сериализованного мешка ячеек в памяти.

5.3.6. Сериализация одной ячейки из мешка ячеек. Точнее, каждая отдельная ячейка $c = c_i$ сериализуется следующим образом, при условии, *что s* кратна восьми (обычно $s = 8, 16, 24$ или 32):

1. Два дескрипторных байта d_1 и d_2 вычисляются аналогично [4, 3.1.4] путем установки $d_1 = r + 8s + 16h + 32l$ и $d_2 = \lfloor b/8 \rfloor + \lceil b/8 \rceil$, где:

- $0 \le r \le 4$ — количество клеточных ссылок, присутствующих в *ячейке c; если c* отсутствует в пакете сериализуемых ячеек и представлен только его хэшами, то $r = 7$.[38]
- $0 \le b \le 1023$ — количество битов данных в ячейке *c.*
- $0 \le l \le 3$ — уровень клетки *c* (см. [4, 3.1.3]).
- $s = 1$ для экзотических клеток и $s = 0$ для обычных клеток.
- $h = 1$, если хэши ячейки явно включены в сериализацию; в противном случае $h = 0$. (Когда $r = 7$, мы всегда должны иметь $h = 1$.)

Для отсутствующих ячеек (т.е. внешних ссылок) присутствует только d_1, всегда равный 23 + 32l.

2. Два байта d_1 и d_2 (если $r < 7$) или один байт d_1 (если $r = 7$) начинают сериализацию ячейки *c.*

3. Если $h = 1$, сериализация продолжается $l + 1$ 32-байтовыми более высокими хэшами *c* (см. [4, 3.1.6]): $\text{Hash}_1(c),\ldots, \text{Hash}_{l+1}(c) = \text{Hash}_\infty(c)$.

4. После этого байты данных $\lceil b/8 \rceil$ сериализуются, путем разделения битов данных b на 8-битные группы и интерпретации каждой группы как целое число с большим порядком байтов в диапазоне 0…255. Если b не делится на 8, то биты данных сначала дополняются одним двоичным файлом 1 и до шести двоичных 0, чтобы число битов данных делилось на восемь.[39]

[38] Обратите внимание, что эти «отсутствующие ячейки» отличаются от библиотечных ссылочных и внешних ссылочных ячеек, которые являются разновидностями экзотических ячеек (ср. [4, 3.1.7]). Отсутствующие клетки, напротив, вводятся только с целью сериализации неполных мешков клеток и никогда не могут быть обработаны TVM.

[39] Обратите внимание, что экзотические ячейки (с $s = 1$) всегда имеют $b \ge 8$, причем тип ячейки закодирован в первых восьми битах данных (ср. [4, 3.1.7]).

5. Наконец, ссылки r-ячеек на ячейки $c_{j1},\ldots, c_{jr}$ кодируются с помощью r *s-бит больших порядковых чисел* $j_1,\ldots, j_r$.[40]

5.3.7. Классификация схем сериализации для мешков ячеек. Схема сериализации для пакета ячеек должна указывать следующие параметры:

- 4-байтовое магическое число, предшествующее сериализации.
- Количество битов, используемых для представления индексов ячеек. Обычно s кратно восьми (например, $8, 16, 24$ или 32).
- Число битов t, используемых для представления смещений сериализаций ячеек (см. 5.3.5). Обычно t также кратно восьми.
- Флаг, указывающий, присутствует ли индекс со смещениями $L_0,\ldots, L_n$ сериализаций ячеек. Этот флаг можно комбинировать с t, установив $t = 0$ при отсутствии индекса.
- Флаг, указывающий, добавляется ли к нему CRC32-C всей сериализации в целях проверки целостности.

5.3.8. Поля, присутствующие при сериализации мешка ячеек. Помимо значений, перечисленных в разделе 5.3.7, фиксированных выбором схемы сериализации для мешков ячеек, сериализация конкретного мешка ячеек должна указывать следующие параметры:

- Общее количество ячеек n, присутствующих в сериализации.
- Количество «корневых клеток» $k \le n$, присутствующих в сериализации. Сами корневые клетки $c_0,\ldots, c_{k-1}$. *Ожидается, что все другие клетки, присутствующие в мешке клеток, будут доступны с помощью цепочек ссылок, начиная с корневых клеток.*

[40] Если мешок клеток не является полным, некоторые из этих клеточных ссылок могут относиться к клеткам c', отсутствующим в сумке клеток. В этом случае специальные «отсутствующие ячейки» с $r = 7$ включаются в мешок клеток и им присваиваются некоторые индексы j. *Эти индексы затем используются для представления ссылок на отсутствующие ячейки.*

- Число «отсутствующих клеток» $l \leq n - k$, *которые представляют собой клетки, которые фактически отсутствуют в этом мешке клеток, но относятся к нему. Сами отсутствующие клетки представлены* c_{n-l},..., c_{n-1}, *и только эти клетки могут (и тоже должны) иметь* $r = 7$. Полные мешки клеток имеют $l = 0$.

- Общая длина в байтах L_n сериализации всех ячеек. Если индекс присутствует, L_n может не храниться явно, так как он может быть восстановлен в качестве последней записи индекса.

5.3.9. Схема TL-B для сериализации мешков ячеек. Несколько конструкторов TL-B могут быть использованы для сериализации пакетов ячеек в октетные (т.е. 8-битные байтовые) последовательности. Единственный, который в настоящее время используется для сериализации новых пакетов клеток, это

```
serialized_boc#b5ee9c72 has_idx:(## 1) has_crc32c:(## 1)
  has_cache_bits:(## 1) flags:(## 2) { flags = 0 }
  size:(## 3) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots >= 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  root_list:(roots * ##(size * 8))
  index:has_idx? (ячейки * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:has_crc32c?uint32
  = BagOfCells;
```

Полевые ячейки — *n, корни — k, отсутствующие — l, а tot_cells_size —* L_n (суммарный размер сериализации всех ячеек в байтах). При наличии индекса параметры $s/8$ и $t/8$ сериализуются отдельно как размер и off_bytes соответственно, и устанавливается флаг has_idx. Сам индекс содержится в индексе, присутствует только в том случае, если задан has_idx. Поле root_list содержит (отсчитываемые от нуля) индексы корневых узлов ячейки.

Два старых конструктора по-прежнему поддерживаются в функциях десериализации bag-of-cells:

```
serialized_boc_idx#68ff65f3 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  = BagOfCells;

serialized_boc_idx_crc32c#acc3a728 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:uint32 = BagOfCells;
```

5.3.10. Хранение скомпилированного TVM-кода в файлах. Обратите внимание, что приведенная выше процедура сериализации пакетов ячеек может быть использована для сериализации скомпилированных смарт-контрактов и другого кода TVM. Необходимо определить конструктор TL-B, подобный следующему:

```
compiled_smart_contract
  compiled_at:uint32 code:^Cell data:^Cell
  description:(Maybe^TinyString)
  _:^[ source_file:(Maybe^ TinyString)
       compiler_version:(Maybe^ TinyString) ]
```

```
= CompiledSmartContract;

tiny_string#_ len:(#<= 126) str:(len * [ uint8 ]) = TinyString;
```

Затем скомпилированный смарт-контракт может быть представлен значением типа *CompiledSmartContract, преобразован в дерево ячеек, а затем в мешок ячеек, а затем сериализован с помощью одного из конструкторов, перечисленных в 5.3.9. Полученная строка октета может быть затем записана в файл с суффиксом .tvc («смарт-контракт TVM»), и этот файл может быть использован для распространения скомпилированного смарт-контракта, загрузки его в приложение кошелька для развертывания в блокчейне TON и так далее.*

5.3.11. Хэши Меркла для октетической струны. В некоторых случаях мы должны определить хэш-хэш Меркла $\text{Hash}_M(s)$ произвольной строки октета s длиной $|s|$. Мы делаем это следующим образом:

- Если $|s| \le 256$ октетов, то хэш Merkle s — это просто его sha256:

$$\text{Hash}_M(s) := \text{sha256}(s) \qquad \text{если} \quad |s| \le 256. \tag{19}$$

- Если $|s| > 256$, пусть $n = 2^k$ будет наибольшей мощностью двух меньше $|c|$ (т.е. $k := \lfloor \log_2(|s| - 1) \rfloor, n := 2^k$). Если s' — префикс s длины n, *а* s'' — суффикс s длины $|s| - n$, *так что* s — конкатенация $s'.s''$ из s' и s'', *определим*

$$\text{Hash}_M(s) := \text{sha256}\,(\text{INT}_{64}(|s|).\ \text{Hash}_M(s').\ \text{Hash}_M(s'')) \tag{20}$$

  Другими словами, мы объединяем 64-битное представление биг-эндиана $|s|$ и рекурсивно вычисленные хэши Меркла s' и s'' *и вычислить sha256 результирующей строки.*

Можно проверить, что $\text{Hash}_M(s) = \text{Hash}_M(t)$ для строк октета s и t длиной менее $264 - 256$ подразумевает $s = t$, если не найдено хэш-коллизия для sha256.

5.3.12. Хеш сериализации блока. Конструкция раздела 5.3.11 применяется, в частности, к сериализации мешка ячеек, представляющих собой неподписанный шардчейн или мастерчейн. Валидаторы подписывают не только хэш представления беззнакового блока, но и «хэш сериализации» беззнакового блока, определяемый как Hash_M сериализации беззнакового блока. Таким образом, валидаторы удостоверяют, что эта строка октета действительно является сериализацией соответствующего блока.

Ссылки

[1] Daniel J. Bernstein, Curve25519: New Diffie–Hellman Speed Records (2006), in: M. Yung, Ye. Dodis, A. Kiayas et al, Public Key Cryptography, Lecture Notes in Computer Science 3958, pp. 207–228. Available at https://cr.yp.to/ecdh/curve25519-20060209.pdf.

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange et al., Highspeed high-security signatures (2012), Journal of Cryptographic Engineering 2 (2), pp. 77–89. Available at https://ed25519.cr.yp.to/ ed25519-20110926.pdf.

[3] N. Durov, Telegram Open Network, 2017.

[4] N. Durov, Telegram Open Network Virtual Machine, 2018.

A Криптография эллиптических кривых

Это приложение содержит формальное описание криптографии эллиптической кривой, используемой в настоящее время в TON, особенно в блокчейне TON и сети TON.

TON использует две формы криптографии эллиптических кривых: Ed25519 используется для криптографических сигнатур Шнорра, а Curve25519 используется для асимметричной криптографии. Эти кривые используются стандартным способом (как определено в оригинальных статьях [1] и [2] Д. Бернштейна и RFC 7748 и 8032); однако некоторые детали сериализации, характерные для TON, должны быть объяснены. Одна из уникальных адаптаций этих кривых для TON заключается в том, что TON поддерживает автоматическое преобразование ключей Ed25519 в ключи Curve25519, так что одни и те же ключи могут использоваться для сигнатур и для асимметричной криптографии.

A.1 Эллиптические кривые

Некоторые общие факты об эллиптических кривых над конечными полями, относящиеся к криптографии эллиптических кривых, собраны в этом разделе.

A.1.1. Конечные поля. Рассмотрим эллиптические кривые над конечными полями. Для целей алгоритмов Curve25519 и Ed25519 мы будем в основном заниматься эллиптическими кривыми над конечным простым полем $k := F_p$ остатков по модулю p, *где* $p = 2255 - 19$ — простое число, и над конечными расширениями F_q F_p, особенно квадратичным расширением F_{p2}.[41]

A.1.2. Эллиптические кривые. *Эллиптическая кривая* $E = (\mathrm{E},\mathrm{O})$ над полем k представляет собой геометрически интегральную гладкую проективную

[41] Арифметический модуль p для модуля p вблизи степени два может быть реализован очень эффективно. С другой стороны, остатки по модулю $2255 - 19$ могут быть представлены 255-битными целыми числами. Именно по этой причине именно это значение p было выбрано Д. Бернштейном.

кривую E/k рода $g = 1$, наряду с отмеченной *k-рациональной точкой* $O \in E(k)$. *Известно, что эллиптическая кривая* E над полем k может быть представлена в (обобщенной) форме Вейерштрасса:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \text{ для некоторых } a_1,\ldots, a_6 \in k. \quad (21)$$

Точнее, это только аффинная часть эллиптической кривой, записанная в координатах (x,y). Для любого расширения поля K из k, $E(K)$ состоит из всех решений $(x,y) \in K^2$ уравнения (21), называемых конечными *точками E(K), вместе с точкой в бесконечности, которая является отмеченной точкой O.*

А.1.3. Форма Вейерштрасса в однородных координатах. В однородных координатах $[X : Y : Z]$, *(21) соответствует*

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \quad (22)$$

Когда $Z \neq 0$, мы можем установить $x := X/Z$, $y := Y/Z$ *и получить решение* (x,y) из (21) (т.е. конечную точку *E). С другой стороны, единственным решением (вплоть до пропорциональности) (22) с* $Z = 0$ является $[0 : 1 : 0]$; это точка в бесконечности *O.*

А.1.4. Стандартная форма Вейерштрасса. Когда характеристический символ поля $k \neq 2, 3$, форма Вейерштрасса (21) или (22) может быть упрощена с помощью линейных *преобразований* $y' := y + a_1x/2 + a_3/2$, $x' := x + a_2/3$, *таким образом получая* $a_1 = a_3 = a_2 = 0$ и получая

$$y^2 = x^3 + a_4x + a_6 \quad (23)$$

и

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \quad (24)$$

Такое уравнение определяет эллиптическую кривую тогда и только тогда, когда кубический многочлен
$P(x) := x^3 + a_4x + a_6$ не имеет нескольких корней, т.е. если дискриминант $D := -4a_4^3 - 27a_6^2$ не равен нулю.

А.1.5. Сложение точек на эллиптической кривой *E. Пусть K* — расширение поля *k, а E* = (E,O) — любая эллиптическая кривая в форме Вейерштрасса, определённая над *k. Тогда любая прямая l* ⊂ P^2_K пересекает эллиптическую кривую *E(K)* (которая является базовым изменением кривой *E* на поле *K, т. е. кривой, определяемой теми же уравнениями над большим полем K) ровно в трех точках P, Q, R, рассматриваемых с кратностью. Мы определяем сложение точек* на эллиптической кривой *E* (или, скорее, сложение ее *K-значных точек E(K)), требуя, чтобы*

$P + Q + R = O$ всякий раз, когда $\{P,Q,R\} = l \cap E$ для некоторой линии $l \subset P^2_K$. (25)

Хорошо известно, что это требование определяет уникальный коммутативный закон $[+] : E \times_k E \to E$ на точках эллиптической кривой *E, имея O* в качестве нейтрального элемента. Когда эллиптическая кривая *E* представлена формой Вейерштрасса (21), можно записать явные формулы, выражающие координаты x_{P+Q}, y_{P+Q} суммы *P* +Q двух *K-значных точек P, Q* ∈ *E(K)* эллиптической кривой *E* как рациональные функции координат x_P, y_P , $x_Q, y_Q \in K$ точек *P* и *Q* и коэффициентов $a_i \in k$ (21).

А.1.6. Карты мощности. Поскольку *E(K)* является абелевой группой, можно определить *кратные* или *степени* [n]X для любой точки $X \in E(K)$ и любого целого числа $n \in Z$. Если $n = 0$, то $[0]X = O$; *если* $n > 0$, то $[n]X = [n-1]X + X$; *если* $n < 0$, то $[n]X = -[-n]X$. Карта $[n] = [n]_E : E \to E$ для $n \neq 0$ является *изогенией, что означает, что она является непостоянным гомоморфизмом для группового закона E:*

$$[n](P + Q) = [n]P + [n]Q \quad \text{для любых } P, Q \in E(K) \text{ и } n \in Z. \tag{26}$$

В частности, $[-1]_E : E \to E$, $P \to 7 -P$, является инволютивным автоморфизмом эллиптической кривой *E. Если E* находится в форме Вейерштрасса, $[-1]_E$ карты $(x,y) \to (x,-y)$, а две точки $P, Q \in E(F_q)$ имеют *равные x-координаты тогда и только тогда, когда* $Q = \pm P$.

А.1.7. Порядок группы рациональных точек *E. Пусть E* — эллиптическая кривая, определённая над конечным базовым полем *k, и пусть* $K = F_q$ —

конечное расширение k. *Тогда* $E(F_q)$ является конечной абелевой группой. По известному результату Хассе порядок n этой группы не слишком далек от q:

$$n = |\,E(F_q)| = q - t + 1, \text{ где } t^2 \le 4q, \text{ т.е. } |t| \le 2\sqrt{q}. \qquad (27)$$

Нас больше всего заинтересует случай $K = k = F_p$, где $q = p$ — простое число.

А.1.8. Циклические подгруппы большого простого порядка. Криптография эллиптических кривых обычно выполняется с использованием эллиптических кривых, которые допускают (обязательно циклическую) подгруппу $C \subset E(F_q)$ простого порядка e. *Эквивалентно, может быть задана рациональная точка* $G \in E(F_q)$ простого порядка e; тогда C может быть восстановлена как циклическая подгруппа $\langle G \rangle$, порожденная G. *Чтобы убедиться, что точка* $G \in E(F_q)$ порождает циклическую группу простого порядка e, *можно проверить, что* $G \neq O$, *но* $[e]G = O$.

Согласно теореме Лежандра, e обязательно является делителем порядка $n = |\,E(F_q)|$ конечной абелевой группы $E(F_q)$:

$$n = |\,E(F_q)| = c\,e \text{ для некоторого целого числа } c \ge 1 \qquad (28)$$

Целое число c называется *кофактором; обычно хочется, чтобы кофактор был как можно меньше, чтобы сделать* $e = n/c$ как можно большим. Напомним, что n всегда имеет тот же порядок величины, что $и$ q на (27), поэтому его нельзя сильно изменить, изменяя E после того, как q зафиксирован.

А.1.9. Данные для криптографии эллиптических кривых. Чтобы определить конкретную криптографию эллиптической кривой, необходимо зафиксировать конечное базовое поле F_q (если $q = p$ является простым числом, достаточно зафиксировать простое p), *эллиптическую кривую* E/F_q (обычно представленную коэффициентами ее формы Вейерштрасса (23) или (21)), базовую точку O (которая обычно является точкой бесконечности эллиптической кривой, записанной в форме Вейерштрасса), и генератор $G \in E(F_q)$ (обычно определяется его координатами (x,y)

относительно уравнения эллиптической кривой) циклической подгруппы большого простого порядка *e. Простое число e* и кофактор *c* обычно также являются частью данных эллиптической криптографии.

А.1.10. Основные операции криптографии эллиптических кривых. Криптография эллиптических кривых обычно имеет дело с фиксированной циклической подгруппой C большого простого порядка e внутри группы точек эллиптической кривой E над конечным полем F_q. Генератор G из C обычно фиксирован. Обычно предполагается, что, учитывая точку X из C, *нельзя найти ее «дискретное логарифмическое основание G» (т. е. остаток n* по модулю *e* такой, что $X = [n]G$) быстрее, чем в *операциях* $O(\sqrt{e})$. Наиболее важными операциями, используемыми в криптографии эллиптических кривых, являются сложение точек из $C \subset E(F_q)$ и вычисление их степеней, или кратных.

А.1.11. Закрытые и открытые ключи для криптографии эллиптических кривых. Обычно закрытым ключом для криптографии эллиптических кривых, описываемым данными, перечисленными в А.1.9, является «случайное» целое число $0 < a < e$, *называемое секретной экспонентой, а соответствующим открытым ключом является точка* $A := [a]G$ (или просто ее x*coordinate* x_A*), соответствующим образом сериализованная.*

А.1.12. Кривые Монтгомери. Эллиптические кривые с конкретным уравнением Вейерштрасса

$$y^2 = x^3 + Ax^2 + x \quad \text{, где } A = 4a - 2 \text{ для некоторых } \in k,\ a \neq 6,\ a \neq 1 \qquad (29)$$

называются *кривыми Монтгомери. Они обладают удобным свойством, что* $x_{P+Q}x_{P-Q}$ может быть выражена как простая рациональная функция x_P и x_Q*:*

$$x_{P+Q}x_{P-Q} = \left(\frac{x_P x_Q - A}{x_P - x_Q}\right)^2 \qquad (30)$$

Это означает, что x_{P+Q} может быть вычислен при условии, *что* x_{P-Q}, x_P и x_Q известный. В частности, если *известны* x_P, $x_{[n]P}$ и $x_{[n+1]P}$, то $x_{[2n]P}$, $x_{[2n+1]P}$ и $x_{[2n+2]P}$ могут быть вычислены. Используя двоичное представление 0

$< n < 2^s$, можно вычислить $x_{[\lfloor n/2^{s-i} \rfloor]P}$, $x_{[\lfloor n/2^{s-i} \rfloor+1]P}$ для $i = 0, 1,\ldots, s$, *получив таким образом* $x_{[n]P}$ (этот алгоритм для быстрого вычисления $x_{[n]P}$, начиная с x_P на кривых Монтгомери, называется лестницей *Монтгомери). Следовательно, мы видим важность кривых Монтгомери для криптографии эллиптических кривых.*

A.2 Криптография Curve25519

В этом разделе описывается известная криптография Curve25519, предложенная Даниэлем Бернштейном [1], и ее использование в TON.

A.2.1. Curve25519. *Curve25519* определяется как кривая Монтгомери y^2 $= x^3 + Ax^2 + x$ над F_p, где $p = 2^{255} - 19$ и $A = 486662$. (31)

Порядок этой кривой равен $8e$, где e — простое число, а $c = 8$ — кофактор. Циклическая подгруппа порядка ′ порождается точкой G с $x_G = 9$ (это определяет G до знака y_G, *что неважно). Порядок квадратичного скручивания* $2y^2 = x^3 + Ax^2 + x$ Curve25519 равен $4\ e$‘ для другого простого числа *e’.*[42]

A.2.2. Параметры Curve25519. Параметры Curve25519 следующие:

- Базовое поле: Простое конечное поле F_p для $p = 2255 - 19$.
- Уравнение: $y^2 = x^3 + Ax^2 + x$ для $A = 486662$.
- Базовая точка *G: характеризуется* $x_G = 9$ (девять — наименьшее положительное целое число *x-координат генератора подгруппы большого простого порядка* $E(F_p)$*).*
- Порядок $E(F_p)$*:*

$$|E(F_p)| = p - t + 1 = 8e, \quad \text{где} \tag{32}$$

[42] Собственно, Д. Бернштейн выбрал $A = 486662$ потому что это наименьшее положительное целое число.

$A \equiv 2 \pmod 4$ таким образом, что и соответствующая кривая Монтгомери (31) над F_p для $p = 2^{255} - 19$, и квадратичная кривая этой кривой имеют небольшие кофакторы. Такое расположение позволяет избежать необходимости проверять, определяет ли *x-координата* $x_P \in F_p$ точки P точку $(x_P, y_P) \in F^2_p$, лежащую на самой кривой Монтгомери или на ее квадратичном скручивании.

$e = 2^{252} + 27742317777372353535851937790883648493$ является простым. (33)

- Порядок $\tilde{E}(F_p)$, *где* $\tilde{E}$ — квадратичное скручивание E:

$$|\tilde{E}(F_p)| = p + t + 1 = 2p + 2 - 8e = 4e', \quad \text{где} \tag{34}$$

$e' = 2^{253} - 55484635554744707071703875581767296995$ является простым. (35)

A.2.3. Закрытый и открытый ключи для стандартного Curve25519 cryptography. Закрытый ключ для криптографии Curve25519 обычно определяется как секретная *экспонента a, в то время как соответствующий открытый ключ — x_A, координата x точки* $A := [a]G$. Обычно этого достаточно для выполнения ECDH (эллиптическая кривая Обмена ключами Диффи-Хеллмана) и асимметричной криптографии эллиптических кривых, а именно:

Если сторона хочет отправить сообщение M другой стороне, которая имеет открытый ключ x_A (и закрытый ключ *a), выполняются следующие вычисления. Генерируется одноразовая случайная секретная экспонента* b, а $x_B := x_{[b]G}$ и $x_{[b]A}$ вычисляются с помощью лестницы Монтгомери. После этого сообщение M шифруется симметричным шифром, таким как AES, используя 256-битный «общий секрет» $S := x_{[b]A}$ в качестве ключа, а 256-битное целое число («одноразовый открытый ключ») x_B предваряется зашифрованному сообщению. Как только сторона с открытым ключом x_A получает это сообщение, она может вычислять $x_{[a]B}$, начиная с x_B (передается с зашифрованным сообщением) и закрытого *ключа a. Поскольку* $x_{[a]B} = x_{[ab]G} = x_{[b]A} = S$, *принимающая сторона восстанавливает общий секрет* S и может расшифровать оставшуюся часть сообщения.

A.2.4. Открытые и закрытые ключи для криптографии TON Curve25519. TON использует другую форму для открытых и закрытых ключей криптографии Curve25519, заимствованную из криптографии Ed25519.

Закрытый ключ для криптографии TON Curve25519 — это просто случайная 256-битная строка k. *Он используется для вычисления sha512(k), взятия первых 256 бит результата,* интерпретации их как 256-битного целого числа a с младшим порядком байтов *a, очистки битов* 0, 1, 2 и 255 из *a и установки бита* 254 таким образом, чтобы получить значение $2^{254} \le a < 2^{255}$, делимое на восемь. Полученное таким образом

значение a является *секретной экспонентой*, *соответствующей k; между тем, оставшиеся 256 бит sha512(k)* составляют *секретную соль k".*

Открытый ключ, *соответствующий k или секретной экспоненте a, является просто координатой* x_A точки $A := [a]G$. После *вычисления a* и x_A они используются точно так же, как и в A.2.3. В частности, если x_A необходимо сериализовать, он сериализуется в 32 октета как беззнаковое 256-битное целое число с младшим порядком байтов.

A.2.5. Curve25519 используется в сети TON. Обратите внимание, что асимметричная криптография Curve25519, описанная в A.2.4, широко используется сетью TON, особенно протоколом ADNL (Abstract Datagram Network Layer). Тем не менее, TON Blockchain нуждается в криптографии эллиптических кривых в основном для подписей. Для этого используются подписи Ed25519, описанные в следующем разделе.

A.3 Ed25519 криптография

Криптография Ed25519 широко используется для быстрых криптографических подписей как блокчейном TON, так и сетью TON. В этом разделе описывается вариант криптографии Ed25519, используемый TON. Важным отличием от стандартных подходов (определяемых D. Bernstein et al. в [2]) является то, что TON обеспечивает автоматическое преобразование закрытых и публичных ключей Ed25519 в ключи Curve25519, так что одни и те же ключи могут использоваться как для шифрования/расшифровки, так и для подписи сообщений.

A.3.1. Скрученные кривые Эдвардса. *Скрученная кривая Эдвардса* $E_{a,d}$ с параметрами $a \neq 0$ и $d \neq 0$,a над полем k задается уравнением

$$E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2 \qquad \text{над } k \tag{36}$$

Если $a = 1$, это уравнение определяет (раскрученную) кривую Эдвардса. Точка *O(0,1)* обычно выбирается в качестве обозначенной точки $E_{a,d}$.

A.3.2. Искривленные кривые Эдвардса бирационально эквивалентны кривым Монтгомери. Искривленная кривая Эдвардса $E_{a,d}$ бирационально эквивалентна эллиптической кривой Монтгомери

$$M_A : v^2 = u^3 + Au^2 + u \tag{37}$$

где $A = 2(a + d)/(a - d)$ и $d/a = (A - 2)/(A + 2)$. Бирациональная эквивалентность $\varphi : E_{a,d} \rightarrow M_A$ и ее обратный $\varphi{-}1$ задаются формулой

$$\phi : (x, y) \mapsto \left(\frac{1+y}{1-y}, \frac{c(1+y)}{x(1-y)}\right) \tag{38}$$

и

$$\phi^{-1} : (u, v) \mapsto \left(\frac{cu}{v}, \frac{u-1}{u+1}\right)$$

$$118 \tag{39}$$

где

$$c = \sqrt{\frac{A+2}{a}} \tag{40}$$

Обратите внимание, что φ преобразует отмеченную *точку* $O(0,1)$ $E_{a,d}$ в отмеченную точку M_A (т.е. ее точку в бесконечности).

А.3.3. Сложение точек на искривленной кривой Эдвардса. Поскольку $E_{a,d}$ бирационально эквивалентен эллиптической кривой M_A, *сложение точек на* M_A может быть перенесено в $E_{a,d}$ путем установки

$$P + Q := \phi^{-1}\big(\phi(P) + \phi(Q)\big) \text{для любого } P, Q \in E_{a,d}(k).(41)$$

Обратите внимание, что отмеченная точка $O(0,1)$ является нейтральным элементом по отношению к этому дополнению, и что $-(x_P, y_P) = (-x_P, y_P)$.

А.3.4. Формулы для добавления точек на искривленной кривой Эдвардса. Координаты x_{P+Q} и y_{P+Q} допускают простые выражения как рациональные функции x_P, y_P, x_Q, y_Q:

$$x_{P+Q} = \frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q} \tag{42}$$

$$y_{P+Q} = \frac{y_P y_Q - a x_P x_Q}{1 - d x_P x_Q y_P y_Q} \tag{43}$$

Эти выражения могут быть эффективно вычислены, особенно если $a = -1$. Именно по этой причине искривленные кривые Эдвардса важны для криптографии быстрых эллиптических кривых.

A.3.5. Ed25519 как скрученная кривая Эдвардса. Ed25519 — закрученная кривая Эдвардса $E_{-1,d}$ над F_p, где $p = 2^{255} - 19$ — то же простое число, что и для Curve25519, а $d = -(A-2)/(A+2) = -121665/121666$, где $A = 486662$ то же самое, что и в уравнении (31):

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2 \quad \text{для } x, y \in F_p,\ p = 2255 - 19. \quad (44)$$

Таким образом, кривая Ed25519 $E_{-1,d}$ бирационно эквивалентна Curve25519 (31), и можно использовать $E_{-1,d}$ и формулы (42)–(43) для сложения точек на Ed25519 или Curve25519, используя (38) и (39) для преобразования точек на Ed25519 в соответствующие точки на Curve25519, и наоборот.

A.3.6. Генератор Ed25519. Генератором Ed25519 является точка G' с $y(G') = 4/5$ и $0 \le x(G') < p$ чётным. Согласно (38), он соответствует точке (u,v) Curve25519 с $u = (1 + 4/5)/(1 - 4/5) = 9$ (т.е. генератору G Curve25519, выбранному в A.2.2). В частности, $G = \varphi(G')$, G' порождает циклическую подгруппу того же большого простого порядка e, заданного в (32), а для любого целого числа a,

$$\varphi([a]G') = [a]G \ . \quad (45)$$

Таким образом, мы можем выполнять вычисления с помощью Curve25519 и его генератора G *или с помощью Ed25519 и генератора* G' *и получать по существу те же результаты.*

A.3.7. Стандартное представление точек на Ed25519. Точка $P(x,y)$ на Ed25519 может быть представлена двумя ее координатами x_P и y_P, остатками по модулю $p = 2^{255} - 19$. В свою очередь, обе эти координаты могут быть представлены беззнаковыми 255- или 256-битными целыми числами $0 \le x_P, y_P < p < 2^{255}$.

Тем не менее, более компактное представление P одним младшим порядком байтов без знака 256-битным целым числом $\tilde{P}$ обычно используется (и также используется TON). А именно, 255 битов нижнего

порядка $P^{\sim}$ содержат y_P, $0 \le y_P < p < 2^{255}$, а бит 255 используется для хранения $x_P \bmod 2$, бита нижнего порядка x_P. Поскольку y_P всегда определяет x_P до знака (т.е. вплоть до замены x_P *на* $p-x_P$), x_P и $p-x_P$ всегда можно отличить по их биту более низкого порядка, p является нечетным.

Если достаточно знать, ±P до знака, можно игнорировать $x_P \bmod 2$ и рассматривать только 255-битное целое число y_P с младшим порядком байтов, произвольно устанавливая бит 255, игнорируя его ранее определенное значение или очищая его.

A.3.8. Закрытый ключ для Ed25519. Закрытый *ключ* для Ed25519 — это просто произвольная 256-битная строка k. *Секретная экспонента* a и *секретная соль* k'' выводятся из k путем вычисления sha512(k), а затем принятия первых 256 бит этого sha512 в качестве представления с младшим порядковым *числом* a (но с очищенными битами 255, 2, 1 и 0 и набором битов 254); последние 256 бит sha512(k) затем составляют k''.

По сути, это та же процедура, что описана в A.2.4, но с заменой Curve25519 на бирационно-эквивалентную кривую Ed25519. (На самом деле все наоборот: эта процедура является стандартной для криптографии эллиптических кривых на основе Ed25519, и TON расширяет процедуру до Curve25519.)

A.3.9. Открытый ключ для Ed25519. *Открытый ключ,* соответствующий закрытому ключу k для Ed25519, является стандартным представлением (ср. A.3.7) точки $A = [a]G'$, где a — секретная экспонента (ср. A.3.8), определяемая закрытым ключом k.

Обратите внимание, что $\varphi(A)$ является открытым ключом для Curve25519, определяемым тем же закрытым ключом k в соответствии с A.2.4 и (45). Таким образом, мы можем конвертировать открытые ключи для Ed25519 в соответствующие открытые ключи для Curve25519, и наоборот. Закрытые ключи вообще не нужно трансформировать.

A.3.10. Криптографические Ed25519-подписи. Если сообщение (строка октета) M должно быть подписано закрытым ключом k, определяющим секретную экспоненту a и секретную соль k'', *выполняются следующие вычисления:*

- r := sha512(k''| M), интерпретируемый как 512-битное целое число с младшим порядком байтов. Здесь $s|t$ обозначает сцепление октетовых струн s и t.
- R := [r]G' — точка на Ed25519.
- $R\tilde{}$ — стандартное представление (ср. А.3.7) точки R в виде 32-октетной струны.
- s := $r + a \cdot$ sha512($R\tilde{}$ | $A\tilde{}$ | M) mod e, *закодированный как 256-битное целое число байтов. Здесь* $A\tilde{}$ является стандартным представлением точки A = [a]G', открытого ключа, соответствующего k.

Сигнатура (Шнорра) представляет собой 64-октетную строку (R,$s\tilde{}$), состоящую из стандартного представления точки R и 256-битного целого числа s.

А.3.11. Проверка подписей Ed25519. Для проверки подписи (R,$s\tilde{}$) сообщения *M, предположительно сделанного владельцем закрытого ключа k,* соответствующего известному открытому *ключу A, выполняются следующие шаги:*

- Точки [s]G' и R + [sha512($R\tilde{}$| $A\tilde{}$| M)]A из Ed25519 вычисляются.
- Если эти два пункта совпадают, подпись правильная.