Catchain Consensus: Конспект

Николай Дуров

19 февраля 2020

Аннотация

Цель этого текста состоит в том, чтобы дать представление о протоколе Catchain Consensus Protocol, византийском отказоустойчивом протоколе (BFT), специально разработанном для генерации и проверки блоков в блокчейне TON [3]. Этот протокол потенциально может быть использован для целей, отличных от генерации блоков в блокчейне proof-of-stake (PoS); однако текущая реализация использует некоторые оптимизации, допустимые только для этой конкретной проблемы.

Содержание

Обзор

Протокол Catchain Consensus основан на протоколе построения оверлейной сети и протоколе оверлейного сетевого вещания TON Network ([3]). Сам протокол Catchain Consensus может быть разложен на два отдельных протокола, один более низкоуровневый и общего назначения (*протокол Catchain*[1]), а другой высокоуровневый *протокол Block Consensus Protocol (BCP),* который использует протокол Catchain. Более высокие уровни в стеке протоколов TON заняты уровнями генерации и проверки блоков; однако все они выполняются по существу локально на одной (логической) машине, при этом проблема достижения консенсуса по вновь сгенерированному блоку делегируется на уровень протокола Catchain.

Вот примерная диаграмма стека протоколов, используемого TON для генерации и распределения блоков, показывающая правильное место протокола Catchain Consensus (или, скорее, его двухкомпонентных протоколов):

- *Верхний уровень:* программное обеспечение для генерации блоков и проверки блоков, логически работающее на автономной логической машине, со всеми входами и выходами, обрабатываемыми протоколами более низкого уровня. Задача этого программного обеспечения состоит в том, чтобы либо сгенерировать новый действительный блок для блокчейна (shardchain или мастерчейн блокчейна TON; ср. [3] для обсуждения shardchain и мастерчейна), либо проверить действительность блока, сгенерированного кем-то другим.
- *(TON) Протокол консенсуса блока:* Достигает (византийского отказоустойчивого) консенсуса по блоку, который должен быть принят в качестве следующего в текущей группе валидаторов для мастерчейна или шардчейна. Этот уровень использует (абстрактный

[1] Первоначальное название этого протокола, используемого на начальном этапе фазы исследований и разработок, было *catch-chain* или *catchchain*, потому что это, по сути, специальный блокчейн, предназначенный *для перехвата всех событий, важных для протокола консенсуса; после того, как он много раз говорил и писал это имя, он постепенно был заключен контракт на «catchain».*

интерфейс) программное обеспечение для генерации и проверки блоков и основывается на протоколе Catchain более низкого уровня. Этот протокол более подробно описан в разделе 3.

- *Протокол Catchain:* обеспечивает безопасную постоянную широковещательную рассылку в оверлейной сети (например, целевую группу валидаторов для определенного shardchain или мастерчейна, предназначенную для генерации, проверки и распространения новых блоков в этом shardchain или masterchain), и обнаруживает попытки «обмана» (нарушения протокола) со стороны некоторых участников. Этот протокол более подробно описан в разделе 2.

- *(TON Network) оверлейный широковещательный протокол:* простой широковещательный протокол наилучшего усилия для оверлейных сетей в сети TON, как описано в [3]. Просто транслирует полученные широковещательные сообщения всем соседям в одной и той же оверлейной сети, которые не получали копию этих сообщений раньше, с минимальными усилиями, направленными на хранение копий недоставленных широковещательных сообщений в течение короткого периода времени.

- *(TON Network) протокол оверлея:* создает оверлейные сети (см. [3]) внутри сети протокола ADNL, управляет списками соседей для этих оверлейных сетей. Каждый участник оверлейной сети отслеживает нескольких соседей в одной оверлейной сети и сохраняет выделенные ADNL-соединения (называемые *«каналами»)* с ними, так что входящие сообщения могут эффективно транслироваться всем соседям с минимальными накладными расходами.

- *Абстрактный протокол datagram Network Layer (ADNL):* Базовый протокол сети TON, который доставляет пакеты (датаграммы) между узлами сети, идентифицированными только 256-битными абстрактными (ADNL) адресами, которые фактически являются криптографическими ключами (или их хэшами).

Этот текст направлен на описание только второго и третьего протокола в этом наборе, а именно протокола консенсуса блока (TON) и протокола Catchain (TON).

Мы хотели бы отметить здесь, что автор этого текста, предоставляя общие рекомендации о том, как этот протокол должен быть разработан (на строках «давайте создадим защищенную BFT систему групповых широковещательных сообщений и запустим соответствующим образом адаптированный простой двухфазный или трехфазный протокол фиксации поверх этой системы») и участвуя в нескольких обсуждениях во время разработки и внедрения протокола, определенно не единственный разработчик этого протокола и особенно его текущей реализации. Это работа нескольких человек.

Несколько слов об эффективности объединенного протокола Catchain Consensus. Во-первых, это настоящий византийский протокол Отказоустойчивости (BFT) в том смысле, что он в конечном итоге достигает консенсуса по действительному следующему блоку блокчейна, даже если некоторые участники (валидаторы) демонстрируют произвольно вредоносное поведение, при условии, что эти злонамеренные участники составляют менее одной трети от общего числа валидаторов. Общеизвестно, что достижение консенсуса BFT невозможно, если хотя бы треть участников являются злонамеренными (ср. [5]), поэтому протокол Catchain Consensus настолько хорош, насколько это теоретически возможно в этом отношении. Во-вторых, когда Catchain Consensus был впервые реализован (в декабре 2018 года) и протестирован на 300 узлах, распределенных по всему миру, он достиг консенсуса по новому блоку за 6 секунд для 300 узлов и за 4–5 секунд для 100 узлов (и за 3 секунды для 10 узлов), даже если некоторые из этих узлов не участвуют или демонстрируют неправильное поведение.[2] Поскольку ожидается, что целевые группы TON Blockchain не будут состоять из более чем ста валидаторов (даже если в общей сложности работает тысяча или десять тысяч валидаторов, только

[2] Когда соотношение вредоносных или неучаствующих или очень медленных валидаторов возрастает до одной трети, протокол демонстрирует изящную деградацию, при этом время консенсуса блока растет очень медленно — скажем, не более чем на полсекунды — до тех пор, пока критическое значение в одну треть не будет почти достигнуто.

сотня из них с наибольшими ставками будет генерировать новые блоки мастерчейна, а остальные будут участвовать только в создании новых блоков шардчейна, каждый блок shardchain генерируется и проверяется 10–30 валидаторами; конечно, все цифры, приведенные здесь, являются параметрами конфигурации (ср. [3] и [4]) и могут быть скорректированы позже консенсусным голосованием валидаторов, если это необходимо), это означает, что TON Blockchain способен генерировать новые блоки один раз в 4–5 секунд, как и планировалось изначально. Это обещание было дополнительно протестировано и, как выяснилось, было выполнено с запуском тестовой сети блокчейна TON через пару месяцев (в марте 2019 года). Таким образом, мы видим, что протокол Catchain Consensus является новым членом постоянно растущего семейства практических протоколов BFT (ср. [2]), хотя он основан на несколько иных принципах.

2 Протокол Catchain

Мы уже объясняли в Обзоре (см. 1), что протокол консенсуса BFT, используемый TON Blockchain для достижения консенсуса по новым блокам блокчейна, состоит из двух протоколов. Здесь мы приводим краткое описание *протокола Catchain*, нижнего рычага этих двух протоколов, который потенциально может быть использован для целей, отличных от консенсуса BFT для блоков. Исходный код протокола Catchcain находится в подкаталоге catchain дерева исходного кода.

2.1. Необходимые условия для запуска протокола Catchain. Основным предварительным условием для запуска (экземпляра) протокола Catchain является упорядоченный список всех узлов, которые участвуют (или которым разрешено участвовать) в этом конкретном экземпляре протокола. Этот список состоит из открытых ключей и ADNL-адресов всех участвующих узлов. Он должен быть предоставлен извне при создании экземпляра протокола Catchain.

2.2. Узлы, участвующие в протоколе консенсуса блока. Для конкретной задачи создания новых блоков для одного из блокчейнов (т.е. мастерчейна или одного из активных шардчейнов) блокчейна TON создается специальная целевая группа, состоящая из нескольких валидаторов. Список членов этой целевой группы используется как для создания частной сети оверлея внутри ADNL (это означает, что единственные узлы, которые могут присоединиться к этой оверлейной сети, — это те, которые явно перечислены при ее создании), так и для запуска соответствующего экземпляра протокола Catchain.

Построение этого списка членов является обязанностью более высоких уровней общего стека протоколов (программное обеспечение для создания и проверки блоков) и, следовательно, не является темой этого текста ([4] было бы более подходящей ссылкой). На данный момент достаточно знать, что этот список является детерминированной функцией текущего (самого последнего) состояния мастерчейна (и особенно текущего значения параметров конфигурации, таких как активный список всех валидаторов, выбранных для создания новых блоков вместе с их соответствующими весами). Поскольку список вычисляется детерминированно, все валидаторы вычисляют одни и те же списки, и, в частности, каждый

валидатор знает, в каких целевых группах (т.е. экземплярах протокола Catchain) он участвует без какой-либо дополнительной необходимости в сетевом общении или согласовании.[3]

2.2.1. Уловы создаются заранее. Фактически, вычисляются не только текущие значения списков, упомянутых выше, но и вычисляются их непосредственно последующие (будущие) значения, так что Catchain обычно создается заранее. Таким образом, он уже действует, когда первый блок должен быть создан новым экземпляром целевой группы проверяющего.

2.3. Блок генезиса и идентификатор catchain. Catchain (т.е. экземпляр протокола Catchain) характеризуется своим блоком *генезиса* или *сообщением генезиса*. Это простая структура данных, содержащая некоторые магические числа, назначение catchain (например, идентификатор shardchain, для которого будут генерироваться блоки, и так *называемый порядковый номер catchain*, также полученный из конфигурации masterchain и используемый для различения последующих экземпляров catchain, генерирующих «тот же» shardchain, но, возможно, с разными участвующими валидаторами), и, самое главное, список всех участвующих узлов (их адреса ADNL и открытые ключи Ed25519, как описано в 2.1). Сам протокол Catchain использует только этот список и хэш sha256 общей структуры данных; этот хэш используется в качестве внутреннего идентификатора catchain, т.е. этого конкретного экземпляра протокола Catchain.

2.3.1. Распределение блока генезиса. Обратите внимание, что блок генезиса не распределяется между участвующими узлами; скорее, он вычисляется независимо каждым участвующим узлом, как описано в 2.2. Поскольку хэш блока genesis используется в качестве идентификатора catchain (т.е. идентификатора конкретного экземпляра протокола Catchain; ср. 2.3), если

[3] Если некоторые валидаторы имеют устаревшее состояние мастерчейна, они могут не вычислять правильные списки целевых групп и участвовать в соответствующих перехватах; в этой связи они рассматриваются как вредоносные или неисправные и не влияют на общую действительность протокола BFT, если менее одной трети всех валидаторов терпят неудачу таким образом.

узел (случайно или намеренно) вычисляет другой блок генезиса, он будет фактически заблокирован от участия в «правильном» экземпляре протокола.

2.3.2. Список узлов, участвующих в catchain. Обратите внимание, что (упорядоченный) список узлов, участвующих в catchain, зафиксирован в блоке genesis и, следовательно, известен всем участникам и однозначно определяется хэшем блока genesis (т.е. идентификатором catchain), при условии отсутствия (известных) коллизий для sha256. Поэтому мы фиксируем количество участвующих узлов N в обсуждении одного конкретного catchain ниже и предполагаем, что узлы пронумерованы от 1 до N (их реальные идентичности можно найти в списке участников, использующих этот индекс в диапазоне $1\ldots N$). Множество всех участников будет обозначаться I; мы предполагаем, что $I = \{1\ldots N\}$.

2.4. Сообщения в catchain. Catchain как группа процессов. Одна перспектива заключается в том, что catchain — это *(распределенная) группа процессов, состоящая из* N известных и фиксированных *(взаимодействующих) процессов* (или узлов в предыдущей терминологии), и эти процессы генерируют *широковещательные сообщения*, которые в конечном итоге транслируются всем членам группы процессов. Множество всех процессов обозначается I; мы обычно предполагаем, что $I = \{1\ldots N\}$. Трансляции, генерируемые каждым процессом, нумеруются, начиная с единицы, поэтому *(*n-я трансляция процесса i будет получать *порядковый номер* или *высоту* n; каждая трансляция должна быть однозначно определена идентификатором или индексом i исходного процесса и его высотой n, чтобы мы могли думать о паре i,n) как естественный идентификатор широковещательного сообщения внутри группы процессов. [4] Ожидается, что трансляции, генерируемые одним и тем же процессом i, будут доставлены в каждый другой процесс в том же порядке, в котором они были созданы, т. е. в порядке возрастания их высоты. В этом отношении catchain очень похож на группу процессов в смысле [1] или [7]. Принципиальное отличие состоит в том, что catchain – это «закаленный» вариант процессной группы, терпимый

[4] В византийской среде улова это не обязательно верно во всех ситуациях.

к возможному византийскому (произвольно злонамеренному) поведению некоторых участников.

2.4.1. Отношение зависимости от сообщений. Можно ввести *отношение зависимости ко* всем сообщениям, транслируемым в группе процессов. Это отношение должно быть строгого частичного порядка $\prec$, со свойством, что $m_{i,k} \prec m_{i,k+1}$, где $m_{i,k}$ обозначает k-е сообщение, транслируемое процессом члена группы с индексом i. Значение $m \prec m'$ заключается в том, что m' *зависит от* m, так что (широковещательное) сообщение m' может быть обработано (членом группы процессов) только в том случае, если m было обработано ранее. Например, если сообщение m' представляет собой реакцию члена группы на другое сообщение m, то естественно установить $m \prec m'$. Если член группы процессов получает сообщение m' до того, как все его зависимости, т. е. сообщения $m \prec m'$, были обработаны (или доставлены по протоколу более высокого уровня), то его обработка (или *доставка*) задерживается до тех пор, пока не будут доставлены все его зависимости.

Мы определили отношение зависимости как строгий частичный порядок, поэтому оно должно быть транзитивным ($m'' \prec m'$ и $m' \prec m$ подразумевают $m'' \prec m$), антисимметричным (максимум один из $m' \prec m$ и $m \prec m'$ может удерживать для любых двух сообщений m и m') и антирефлексивным ($m \prec m$ никогда не держится). Если у нас есть меньший набор «базовых зависимостей» $m' \rightarrow m$, мы можем построить его транзитивное закрытие $\rightarrow+$ и поставить $\prec := \rightarrow+$. Единственное другое требование заключается в том, что каждая трансляция отправителя зависит от всех предыдущих трансляций одного и того же отправителя. Не следует строго предполагать это; однако это предположение вполне естественно и значительно упрощает проектирование системы обмена сообщениями внутри группы процессов, поэтому протокол Catchain делает это предположение.

2.4.2. Набор зависимостей или конуса сообщения. Пусть m — (широковещательное) сообщение внутри группы процессов, как указано выше. Мы говорим, что множество $D_m := \{m' : m' \prec m\}$ является множеством *зависимостей* или *конусом зависимости* сообщения m. Другими словами, D_m является *главным идеалом,* порожденным m в частично упорядоченном конечном множестве всех сообщений. Это именно набор всех сообщений, которые должны быть доставлены до m .

2.4.3. Расширенная зависимость конуса сообщения. Мы также определяем D_{m^+}, расширенный *конус зависимости* m, by $D_m^+ := D_m \cup \{m\}$.

2.4.4. Конусы, или идеалы в отношении $\prec$. В более общем плане мы говорим, что подмножество D сообщений является *конусом,* если оно является идеальным по отношению к зависимости отношения $\prec$, т. е. если $m \in D$ и $m' \prec m$ подразумевают $m' \in D$. Конечно, конус зависимости D_m и расширенный конус зависимости D_m^+ любого сообщения m являются конусами (потому что любой главный идеал в частично упорядоченном множестве является идеалом).

2.4.5. Идентификация конусов с помощью векторного времени. Напомним, что мы предположили, что любое сообщение зависит от всех предыдущих сообщений одного и того же отправителя, т.е. $m_{i,s} \prec m_{i,s+1}$ для любого $i \in I$ и любого $s > 0$, так что $m_{i,s+1}$ существует. Это означает, что любой конус D полностью характеризуется N значениями $\mathrm{Vt}(D)_i$, индексируемыми $i \in I$:

$$\mathrm{Vt}(D)_i := \sup\{s \in \mathbb{N} : m_{i,s} \in D\} = \inf\{s \in \mathbb{N}_0 : m_{i,s+1} \notin D\} \qquad (1)$$

(если сообщение $m_{i,s}$ *не* находится в D, мы устанавливаем $\mathrm{Vt}(D)_i := 0$). Действительно, ясно, что

$$m_{i,s} \in D \Leftrightarrow s \leq \mathrm{Vt}(D)_i \qquad (2)$$

Мы говорим, что вектор $\mathrm{Vt}(D) = (\mathrm{Vt}(D)_i)_{i \in I} \in \mathbb{N}_0^I$ с неотрицательными компонентами $\mathrm{Vt}(D)_i$ является *векторным временем* или *векторной меткой времени,* соответствующей конусу D (ср. [1] или [7] для более подробного обсуждения векторного времени).

2.4.6. Частичный порядок по векторным временным меткам. Вводим частичную ≤ порядка на множестве всех возможных векторных времен $\mathbb{N}_0^I$, которая является произведением обычных порядков на $\mathbb{N}_0$:

$$x = (x_i)_{i \in I} \leq y = (y_i)_{i \in I} \quad \text{если} \quad x_i \leq y_i \quad \text{для всех } i \in I \qquad (3)$$

Непосредственно D ⊂ D' если Vt(D) ≤ Vt(D'); поэтому Vt является строгим сохранением порядка встраивания множества всех конусов, содержащихся в множестве всех сообщений $\mathbb{N}_0^I$.

2.4.7. Векторная метка времени Vt(m) сообщения m. Учитывая любое сообщение m, мы определяем его *векторную метку времени* Vt(m) как Vt(D_m). Другими словами, сообщение m может быть доставлено только после доставки первых сообщений Vt(m)$_j$, сгенерированных процессом j, и это верно для всех $j \in I$.

Если i — отправитель сообщения m, а s — высота сообщения m, так что $m = m_{i,s}$, то Vt(m)$_i$ = $s - 1$. Мы можем определить *скорректированную векторную метку времени* Vt$^+$(m) сообщения m , установив $VT^+(m)_j = VT(m)_j$ для $j \neq i$, $VT^+(m)_i$ = Vt(m)$_i$ + 1 = s. Альтернативно, Vt$^+(m) = \mathrm{VT}(D_m^+)$, где $D_{m^+} := D_m \cup \{m\}$ — *конус расширенной зависимости* m (ср. 2.4.3).

Обратите внимание, что $m' \preceq m$ iff $D_{m'}^+ \subset D_m^+$ если Vt$^+(m') \leq \mathrm{VT}^+(m)$ in $\mathbb{N}_0^I$, где $m' \preceq m$ означает «$m' \prec m$ или $m' = m$». Аналогично, $m' \prec m$ iff $D_{m'}^+ \subset D_m$ если Vt$^+$(m') ≤ Vt(m). Другими словами, *отношение зависимости* $\prec$ *на (некоторые или все) сообщения полностью определяется скорректированными векторными временными метками этих сообщений.*

2.4.8. Использование векторных временных меток для корректной доставки широковещательных сообщений. Векторные временные метки можно использовать (в невизантийских настройках) для правильной доставки сообщений, передаваемых в группе процессов.[5] А именно, предположим, что каждое широковещательное сообщение $m = mi,s$ содержит индекс своего отправителя i и векторную метку времени этого сообщения Vt(m). Тогда каждый получатель j знает, может ли сообщение быть доставлено или

[5] Мы предполагаем, что все широковещательные сообщения в группе процессов являются «причинно-следственными трансляциями» или «cbcast» в терминологии [1], потому что нам нужны только cbcasts для реализации протокола Catchain и консенсуса Catchain.

нет. Для этого j отслеживает конус C_j всех сообщений, доставленных до сих пор, например, поддерживая текущую *метку времени* Vt(j), равную Vt(Cj). Другими словами, Vt(j)$_k$ — это количество сообщений отправителя k, обработанных j до сих пор. Если Vt(m) ≤ Vt(j),то сообщение m доставляется немедленно, а Vt(j) обновляется до $\sup$(Vt(j),Vt+(m)) после этого; это эквивалентно увеличению Vt(j) $_i$ на единицу, где i является первоначальным отправителем сообщения m. Если это условие не выполняется, то m может быть помещен в очередь ожидания до тех пор, пока Vt(j) не станет достаточно большим. Вместо того, чтобы пассивно ждать требуемых трансляций, j может построить список индексов сообщений (i',s'), которые неявно упоминаются в Vt(m) некоторого полученного, но не доставленного сообщения m, и запросить сообщения с этими индексами у соседей, от которых j узнал о m и Vt(m); альтернативная стратегия (фактически используемая текущей реализацией протокола Catchain) заключается в том, чтобы время от времени запрашивать эти сообщения у случайно выбранных соседей. Последняя стратегия проще, поскольку не требует запоминания непосредственных источников всех полученных сообщений (которые в любом случае могут стать недоступными).

2.5. Структура сообщения в catchain. Catchain как мульти-блокчейн. Структура сообщений в catchain немного сложнее, чем описано выше, из-за необходимости поддержки протокола BFT. В частности, векторные временные метки недостаточны в византийской обстановке. Они должны быть дополнены описаниями, основанными на максимальных элементах конуса зависимости (такие описания обычно используются в невизантийских условиях только тогда, когда группа процессов очень велика, так что размеры векторных меток времени становятся непомерно высокими).

2.5.1. Описание колбочек с помощью их максимальных элементов. Альтернативным способом (к использованию векторной метки времени) описания конуса сообщения D является перечисление всех его *максимальных элементов* $\mathrm{Max}(D)$, т.е. элементов $m \in D$, так что $m \prec m'$ не удерживается ни для одного $m' \in D$. Конечно, нужен подходящий способ обращения к сообщениям, не включая их полностью, чтобы это представление было практичным.

2.5.2. Идентификаторы сообщений внутри catchain. Протокол Catchain использует хэши sha256 (соответствующим образом сериализованных) сообщений в качестве их уникальных идентификаторов. Если предположить, что для sha256 нет коллизий (вычисляемых в разумном, например, полиномиальном времени), то сообщение m полностью идентифицируется внутри группы процессов по его хэшу sha256(m).

2.5.3. Заголовки сообщений. Заголовок сообщения $m = m_{i,s}$ внутри catchain (т.е. экземпляр протокола Catchain) всегда содержит индекс i его отправителя, высоту s, идентификатор catchain (т.е. хэш сообщения genesis, ср. 2.3) и множество хэшей максимальных элементов конуса зависимости m, т.е., the set $\{\text{sha256}(m') : m' \in \text{Max}(D_m)\}$. В частности, хэш sha256($m_{i,s-1}$) предыдущего сообщения того же отправителя всегда включается, так как $m_{i,s-1} \in \text{Max}(D_m)$, если $s > 1$; по соображениям производительности в заголовке сообщения есть отдельное поле, содержащее sha256($m_{i,s-1}$). Если $s = 1$, то предыдущего сообщения нет, поэтому вместо него используется хэш сообщения genesis (т.е. идентификатор catchain, ср. 2.3).

Векторная метка времени Vt(m) не включена в заголовок сообщения, однако заголовок неявно определяет Vt(m), поскольку

$$\text{VT}(m) = \sup_{m' \in D_m} \text{VT}^+(m') = \sup_{m' \in \text{Max}(D_m)} \text{VT}^+(m') \tag{4}$$

Обратите внимание, что заголовок сообщения является частью сообщения, и, в частности, хэш сообщения (т.е. идентификатор сообщения) зависит от всех данных, перечисленных в заголовке. Поэтому мы предполагаем, что идентификатор сообщения неявно определяет все зависимости соответствующего сообщения (если для sha256 нет известных коллизий).

2.5.4. Подписи сообщений. Кроме того, каждое сообщение в catchain подписывается его создателем. Поскольку список участвующих узлов (процессов) в catchain известен заранее, и этот список включает в себя открытые ключи всех процессов, эти подписи сообщений могут быть проверены процессом получения сразу после получения сообщения. Если

подпись недействительна, сообщение отбрасывается без дальнейшей обработки.

2.5.5. Шифрование сообщений. Все сообщения в catchain также шифруются перед передачей от узла к его соседу в частной оверлейной сети, лежащей в основе catchain. Однако это шифрование выполняется сетевыми протоколами более низкого уровня (такими как ADNL) и не имеет отношения к обсуждению здесь. Мы хотели бы отметить, что правильное шифрование здесь возможно только потому, что список участвующих процессов включает в себя не только открытые ключи всех процессов, но и их ADNL-адреса (которые фактически являются открытыми ключами шифрования для сетевой передачи).

Обратите внимание, что даже если бы шифрование отсутствовало, это не нарушило бы свойства BFT протокола, потому что подделка сообщения от другого отправителя была бы невозможна из-за подписей. Однако это может привести к утечке информации внешним наблюдателям, что часто нежелательно.

2.5.6. Альтернативная перспектива: уловка в виде мультибейтона. Обратите внимание, что все сообщения, созданные одним и тем же отправителем i в catchain, оказываются имеющими простую «структуру блокчейна», поскольку заголовок $m_{i,s+1}$ содержит хэш sha256($m_{i,s}$) (среди прочих хэшей сообщений от $\mathrm{Max}(D_{mi,s+1})$) предыдущего сообщения отправителя i. Таким образом, каждый процесс *генерирует* простой блокчейн, состоящий из его сообщений, причем каждый «блок» этого блокчейна соответствует одному сообщению и ссылается на предыдущий блок своим хэшем, а иногда включает ссылки на блоки (то есть сообщения) других процессов, упоминая хэши этих блоков в своих блоках. Каждый блок подписывается его создателем. Результирующая структура очень похожа на структуру «асинхронного канала платежей», рассматриваемого в [3, 5], но с N участниками вместо 2.

2.6. Распространение сообщения в catchain. Теперь мы готовы описать распространение сообщения в catchain. Именно:

- (Низкоуровневый) оверлейный сетевой протокол поддерживает список соседей в частной оверлейной сети, лежащей в основе catchain, и предоставляет каналы ADNL каждому из этих соседей. Эта

частная оверлейная сеть имеет тот же список членов (процессов, узлов), что и catchain, а соседи каждого узла образуют (ориентированный) подграф на множестве всех участвующих узлов. Этот (по существу случайный) подграф тесно связан с вероятностью, очень близкой к единице.

- Каждый процесс время от времени генерирует новые сообщения (по мере необходимости протокола более высокого уровня). Эти сообщения дополняются заголовками сообщений catchain, как описано в 2.5.3, подписываются и распространяются на всех известных соседей с помощью каналов ADNL, установленных протоколом наложения.

- В отличие от обычного простого протокола вещания наложения, сообщения, полученные от соседей, не сразу ретранслируются всем другим соседям, которые, как известно, еще не имеют их копии. Вместо этого сначала проверяется подпись, а недопустимые сообщения отбрасываются. Затем сообщение либо доставляется (если все его зависимые сообщения уже доставлены), либо помещается в очередь ожидания. В последнем случае все необходимые сообщения, упомянутые в его заголовке (т.е. набор $\mathrm{Max}(D_m)$), извлекаются из соседа, отправившего это сообщение (кроме того, время от времени предпринимаются попытки загрузить эти недостающие сообщения от случайных соседей). При необходимости этот процесс повторяется рекурсивно до тех пор, пока не будут доставлены некоторые сообщения. Как только сообщение готово к локальной доставке (т. е. все его зависимости уже присутствуют), оно также ретранслируется для всех соседей в оверлейной сети.

- Помимо описанного выше рекурсивного механизма «вытягивания», также используется более быстрый векторный механизм на основе меток времени, так что сообщения могут запрашиваться у соседей по их отправителям и высотам (полученным из векторных временных меток полученных сообщений). А именно, каждый процесс время от времени отправляет случайно выбранному соседу специальный запрос, содержащий текущую векторную метку времени. Этот одноранговый запрос приводит к тому, что его получатель отправляет

обратно все или некоторые сообщения, неизвестные отправителю (судя по их векторным временным меткам).

- Этот более быстрый векторный механизм на основе меток времени может быть отключен для сообщений, исходящих от определенных отправителей, как только обнаруживается «форк», то есть второе сообщение с тем же отправителем i и высотой s, но с другим хэшем, изучается у соседа, например, во время быстрого или медленного процесса «вытягивания». Как только форк, созданный i , обнаружен, соответствующий компонент Vt_i всех последующих векторных временных меток устанавливается на специальное значение ∞ , чтобы указать, что сравнение значений этих компонентов больше не имеет смысла.

- При доставке сообщения (по протоколу более высокого уровня) это сообщение добавляется в конус C обработанных сообщений текущего процесса (и текущая векторная метка времени обновляется соответствующим образом), и все последующие сообщения, генерируемые текущим процессом, будут считаться зависящими от всех сообщений, доставленных до сих пор (даже если это логически не является необходимым с точки зрения протокола более высокого уровня).

- Если множество $\mathrm{Max}(C)$ максимальных элементов конуса обрабатываемых сообщений становится слишком большим (содержит больше элементов, чем определенное количество, заранее зафиксированное сообщением генезиса catchain), то протокол Catchain просит протокол более высокого уровня сгенерировать новое сообщение (пустое, если нет полезной полезной нагрузки). После генерации этого нового сообщения (и немедленной доставки в текущий процесс) C обновляется, а $\mathrm{Max}(C)$ состоит только из одного элемента (нового сообщения). Таким образом, размер $\mathrm{Max}(C)$ и, следовательно, размер заголовка сообщения всегда остаются ограниченными.

- После доставки сообщения m и изменения набора C для включения этого сообщения устанавливается таймер, и после некоторой

небольшой задержки протоколу более высокого уровня предлагается создать новое сообщение (пустое, если это необходимо), чтобы это новое сообщение $m*$ ссылалось на новый C, аналогично процедуре, описанной в предыдущем пункте. Это новое сообщение $m*$ передается всем соседям; поскольку его заголовок содержит $\mathrm{Max}(C)$ для нового C, а $m \in C$, соседи узнают не только о вновь сгенерированном *сообщении* $m*$, но и об исходном полученном *сообщении* m. Если у некоторых соседей еще нет копии m , они потребуют ее (из текущего процесса или нет).

- Все (широковещательные) сообщения, полученные и созданные в catchain, хранятся в специальной локальной базе данных. Это особенно важно для вновь созданных сообщений (см. 3.3.2): если сообщение создано и отправлено соседям, но не сохранено в базе данных (и смыто на диск) до сбоя и перезапуска процесса создания, то после перезапуска может быть создано другое сообщение с тем же отправителем и высотой, что фактически приводит к непроизвольной «вилке».

2.7. Вилки и их профилактика. Можно видеть, что описанная выше мультибейн-структура catchain (со ссылками на другие блоки по их хэшам и с подписями) оставляет очень мало возможностей для «обмана» в консенсусном протоколе, построенном на catchain (т. е. использование catchain в качестве средства для трансляции сообщений внутри группы процессов). Единственная возможность, которая не обнаруживается сразу, состоит в создании двух (или более) разных версий одного и того же сообщения $m_{i,s}$ (скажем, $m'_{i,s}$ и $m''_{i,s}$), и отправке одной версии этого сообщения $m'_{i,s}$ одним одноранговым узлам и другой версии $m''_{i,s}$ другим. Если s минимально (для фиксированного i), то это соответствует *форку* в терминологии блокчейна: два разных следующих блока $m'_{i,s}$ и $m''_{i,s}$ для того же предыдущего блока $m_{i,s-1}$.

Поэтому протокол Catchain заботится о том, чтобы как можно скорее обнаружить вилки и предотвратить их распространение.

2.7.1. Обнаружение вилок. Обнаружение форков простое: если есть два разных блока $m'_{i,s}$ и $m''_{i,s}$ с одним и тем же создателем $i \in I$ и одинаковой высоты $s \geq 1$, и с действительными сигнатурами i, то это форк.

2.7.2. Вилочные доказательства. Подписи блоков в протоколе Catchain создаются таким образом, что создание *доказательств форка* (т.е. доказательство того, что процесс i намеренно создал форк) особенно просто, поскольку это хэш очень маленькой структуры (содержащей магическое число, значения i и s , и хэш оставшейся части сообщения), который фактически подписан. Поэтому в доказательстве вилки требуется только две такие небольшие структуры и две подписи.

2.7.3. Внешнее наказание за создание вилок. Обратите внимание, что внешнее наказание за создание форков catchain может использоваться в контексте генерации блокчейна proof-of-stake. А именно, доказательства форка могут быть представлены специальному смарт-контракту (например, смарт-контракту избирателя TON Blockchain), проверены автоматически, и некоторая часть или вся доля нарушителя может быть конфискована.

2.7.4. Внутренняя обработка вилок. Как только форк (созданный i) обнаружен (другим процессом j), т.е. j узнает о двух разных сообщениях $m_{i,s}$ и $m'_{i,s}$созданных i и имеющих одинаковую высоту s (обычно это происходит при рекурсивной загрузке зависимостей некоторых других сообщений), j начинает игнорировать i и все его последующие послания. Они не принимаются и не транслируются в дальнейшем. Однако сообщения, созданные i до обнаружения форка, могут по-прежнему загружаться, если они упоминаются в сообщениях (блоках), созданных процессами, которые не видели этот форк до ссылки на такие сообщения, созданные i.

2.7.5. Принимать сообщения от «плохого» процесса — это плохо. Кроме того, если процесс i узнает о форке, созданном процессом j, то *я* показываю это своим соседям, создавая новое широковещательное сообщение сервиса, содержащее соответствующее доказательство форка (ср. 2.7.2). Впоследствии, это и все последующие сообщения j не могут напрямую зависеть от каких-либо сообщений известного «плохого» производителя i (но они все равно могут ссылаться на сообщения от другой стороны k, которые прямо или косвенно ссылаются на сообщения i, *если* форк і было известно *в момент создания ссылающегося сообщения*). Если j нарушает это ограничение и создает сообщения с такими недопустимыми ссылками, эти сообщения будут отброшены всеми честными процессами в группе.

2.7.6. Набор «плохих» членов группы является частью внутреннего состояния. Каждый процесс i хранит свою копию набора известных «плохих» процессов в группе, т.е. тех процессов, которые создали хотя бы один форк или нарушили 2.7.5. Этот набор обновляется добавлением в него j, как только *я* узнаю о форке, созданном j (или о нарушении 2.7.5 j); после этого вызывается обратный вызов, предоставляемый протоколом более высокого уровня. Этот набор используется при поступлении нового широковещательного сообщения: если отправитель неисправен, то сообщение игнорируется и отбрасывается.

3 Протокол консенсуса блока

В этом разделе мы объясняем основные принципы работы протокола консенсуса блоков TON (см. 1), который основывается на общем протоколе Catchain (см. 2) для обеспечения протокола BFT, используемого для генерации и проверки новых блоков блокчейна TON. Исходный код протокола TON Block Consensus находится в подкаталоге validator-session дерева исходного кода.

3.1. Внутреннее состояние Протокола консенсуса блока. Протокол о консенсусе блока более высокого уровня вводит новое понятие в catchain: *внутреннее состояние* Протокола консенсуса блоков (BCP), иногда также (не совсем корректно) называемое «внутренним состоянием catchain» или просто *состоянием catchain*. А именно, каждый процесс, $который\ i \in I$ имеет четко определенное внутреннее состояние σ_{Ci} после подмножества сообщений (на самом деле всегда конус зависимости) C_i доставляется протоколом Catchain в протокол более высокого уровня (т.е. в протокол Block Consensus Protocol в данном случае). Кроме того, это состояние $\sigma_{Ci} = \sigma(C_i)$ зависит только от конуса C_i, но не от идентичности процесса $i \in I$, и может быть определено для любого конуса зависимости S (не обязательно конуса C_i доставляемых сообщений для некоторого процесса i в какой-то момент).

3.1.1. Абстрактная структура внутреннего состояния. Начнем с абстрактной структуры внутреннего состояния, используемой BCP; более подробная информация будет предоставлена позднее.

3.1.2. Обновление внутреннего состояния. Протокол Catchain ничего не знает о внутреннем состоянии; он просто вызывает соответствующие обратные вызовы, предоставляемые протоколом более высокого уровня (т. е. BCP) всякий раз, когда $доставляется\ сообщение\ m$. Задача протокола более высокого уровня состоит в том, чтобы вычислить новое состояние $\sigma_{S'}$, начиная с ранее вычисленного состояния σ_S и сообщения m, где $S' = S \cup \{m\}$ (и обязательно $S \supset D_m$, иначе m не мог бы быть доставлен в этот момент).

3.1.3. Рекурсивная формула обновления внутреннего состояния. Абстрактная установка для вычисления σ_S для всех конусов S состоит из трех компонентов:

- Значение $\sigma_\emptyset$ для начального состояния (это значение фактически зависит от блока генезиса catchain; мы игнорируем эту зависимость здесь, потому что мы рассматриваем только один catchain на этом этапе).
- Функция f, вычисляющая состояние σ_{Dm+} из предыдущего состояния σ_{Dm} и вновь доставленного сообщения m:

$$\sigma_{Dm+} = f(\sigma_{Dm}, m) \quad (5)$$

где D_m — конус зависимости сообщения m и $D_m^+ = D_m \cup \{m\}$ его расширенный конус зависимости (см. 2.4.3). В большинстве случаев f действительно удовлетворяет более сильному условию.

$$\sigma_{S \cup \{m\}} = f(\sigma_S, m) \text{ если } S \text{ и } S \cup \{m\} \text{ являются конусами и } m \notin S \quad (6)$$

Однако это более сильное условие не требуется алгоритмом обновления.

- «Функция слияния» g, которая вычисляет $\sigma_{S \cup T}$ из σ_S и σ_T:

$$\sigma_{S \cup T} = g(\sigma_S, \sigma_T) \text{ для любых колбочек } S \text{ и } T \quad (7)$$

(объединение двух конусов всегда является конусом). Эта функция σ применяется алгоритмом обновления только в конкретном случае $T = D_{m^+}$ и $m \notin S$.

3.1.4. Коммутативность и ассоциативность g. Обратите внимание, что (7) (для произвольных конусов S и T) подразумевает ассоциативность и коммутативность g, по крайней мере, когда g применяется к возможным состояниям (значения формы σ_S для некоторого конуса S). В этом отношении g определяет коммутативную моноидную структуру на множестве $\Sigma = \{\sigma_S : S \text{ является конусом}\}$. Обычно g определяется или частично определяется на большем множестве $\tilde{\Sigma}$ значений, подобных

состоянию, и оно может быть коммутативным и ассоциативным на этом большем множестве $\tilde{\Sigma}$, т.е. $g(x,y) = g(y,x)$ и $g(x,g(y,z)) = g(g(x,y),z)$ для $x, y, z \in \tilde{\Sigma}$ (всякий раз, когда определены обе стороны равенства), с $\sigma_\emptyset$ как единица, т.е. $g(x,\sigma_\emptyset) = x = g(\sigma_\emptyset,x)$ для $x \in \tilde{S}$ (при тех же условиях). Однако это свойство, полезное для формального анализа алгоритма консенсуса, не является строго обязательным для алгоритма обновления состояния, поскольку этот алгоритм использует g детерминированным образом для вычисления σ_S.

3.1.5. Коммутативность f. Обратите внимание, что f, если он удовлетворяет более сильному условию (6), также должен проявлять свойство коммутативности.

$$f\big(f(\sigma_S, m), m'\big) = f\big(f(\sigma_S, m'), m\big) \tag{8}$$

всякий раз, когда S является конусом, m и m' являются двумя сообщениями с $D_m \subset S$, $D_{m'} \subset S$, $m \notin S$ и $m' \notin S$, потому что в этом случае $S \cup \{m\}$, $S \cup \{m'\}$ и $S \cup \{m,m'\}$ также являются конусами, и (6) подразумевает, что обе стороны (8) равны $\sigma_{S \cup \{m,m'\}}$. Аналогично пункту 3.1.4, f обычно определяется или частично определяется на произведении большего множества $\tilde{\Sigma}$ значений, подобных состоянию, и набора значений, подобных сообщениям; оно может проявлять свойство «коммутативности» (8) или не проявлять на этом большем множестве. Если это так, это может быть полезно для формального анализа алгоритмов, основанных на σ_S, но это свойство не является строго необходимым.

3.1.6. Алгоритм обновления состояния. Алгоритм обновления состояния (независимо выполняемый каждым процессом i), используемый catchain (фактически BCP более высокого уровня), использует $\sigma_\emptyset$, f и g следующим образом:

- Алгоритм отслеживает все σ_{D_m+} для всех сообщений m, доставленных до сих пор.
- Алгоритм отслеживает σ_{C_i}, где C_i — текущий конус зависимости, т. е. множество всех сообщений m , доставленных (к текущему процессу i). Начальное значение σ_{C_i} равно $\sigma_\emptyset$.

- При доставке нового сообщения m значение σ_{Dm} вычисляется повторным приложением g поскольку $D_m = \bigcup_{m' \in D_m} D^+_{m'} = \bigcup_{m' \in \mathrm{Max}(D_m)} D^+_{m'}$; следовательно, если $\mathrm{Max}(D_m) = \{m'_1, \ldots, m'_k\}$, то

$$\sigma_{D_m} = g\Big(\ldots g\big(g(\sigma_{D^+_{m'_1}}, \sigma_{D^+_{m'_2}}), \sigma_{D^+_{m'_3}}\big), \ldots \sigma_{D^+_{m'_k}}\Big) \quad . \qquad (9)$$

  Множество $\mathrm{Max}(D_m)$ явно указано в заголовке сообщения m в некотором фиксированном порядке m'_1, ..., m'_k; приведенная выше формула применяется относительно этого порядка (поэтому вычисление D_m является детерминированным). Первым элементом в этом списке всегда является предыдущее сообщение отправителя m, т.е. если $m = m_{i,s+1}$, то $m'_1 = m_{i,s}$.

- После этого значение σ_{Dm+} вычисляется приложением f: $\sigma_{Dm+} = f(\sigma_{Dm}, m)$. Это значение запоминается для будущего использования.

- Наконец, когда новое сообщение m доставляется текущему процессу i, обновляясь таким образом, $C_i \text{ to } C'_i := C_i \cup \{m\}$ алгоритм использует вычисляемое значение $\sigma Dm+$ для обновления текущего состояния.

$$\sigma_{Ci'} = g(\sigma_{Ci}, \sigma_{Dm+}) \qquad (10)$$

  Это состояние, однако, является «виртуальным» в том смысле, что оно может быть немного изменено позже (особенно если g не является коммутативным). Тем не менее, он используется для принятия некоторых важных решений алгоритмом более высокого уровня (BCP).

- Как только новое сообщение m генерируется и локально доставляется, так что C_i становится равным D_{m^+}, ранее вычисленное значение σ_{Ci} отбрасывается и заменяется σ_{Dm+}, вычисляемым в соответствии с общим алгоритмом, описанным выше. Если g не является коммутативным или не ассоциативным (например, может случиться так, что $g(x,y)$ и $g(y,x)$ являются разными, но эквивалентными представлениями одного и того же состояния), то это

может привести к небольшому изменению текущего «виртуального» состояния процесса i.

- Если протокол нижнего уровня (catchain) сообщает протоколу более высокого уровня, что определенный процесс $j \notin i$ является «плохим» (т.е . j создает форк, ср. 2.7.6, или сознательно одобрил форк другим процессом, ср. 2.7.5), то текущее (виртуальное) состояние σ_{Ci} пересчитывается с нуля с использованием нового набора $C'_i = \bigcup_{m \in C_i,\, m,}$ созданного «хорошим» процессом k D^+_mи функцией «слияние» g применяется к множеству σ_{Dm+}, где m проходит через множество последних сообщений процессов, заведомо хороших (или через множество максимальных элементов этого множества). Следующее созданное исходящее сообщение будет зависеть только от сообщений отC'_i.

3.1.7. Необходимость знать внутреннее состояние других процессов. Формула (9) подразумевает, что процесс i также должен отслеживать σ_{Dm+} для всех сообщений m, созданных этим процессом или нет. Однако это возможно, поскольку эти внутренние состояния также вычисляются соответствующими приложениями алгоритма обновления. Поэтому BCP вычисляет и запоминает все σ_{Dm+}.

3.1.8. Достаточно функции f. Обратите внимание, что алгоритм обновления применяет g только к вычислениям σ_S□$_{Dm+}$ = $g(\sigma_S,\sigma_{Dm+})$, когда S является конусом, содержащим D_m, но не содержащим m. Таким образом, каждое фактическое применение g можно было бы заменить приложением f, удовлетворяющим расширенному свойству (6):

$$\sigma_S\square_{Dm+} = g(\sigma_S,\sigma_{\mathrm{Dm+}}) = f(\sigma_S,m) \qquad (11)$$

Однако алгоритм обновления не использует эту «оптимизацию», потому что он отключит более важные оптимизации, описанные ниже в 3.2.4 и 3.2.5.

3.2. Структура внутреннего государства. Структура внутреннего состояния оптимизирована таким образом, чтобы сделать *переходную функцию* f из (5) и *функцию слияния* g из (7) максимально эффективно вычислимыми, предпочтительно без необходимости потенциально неограниченной

рекурсии (только некоторые циклы). Это мотивирует включение во внутреннее состояние дополнительных компонентов (даже если эти компоненты вычисляются из остальной части внутреннего состояния), которые также должны храниться и обновляться. Этот процесс включения дополнительных компонентов аналогичен тому, который используется при решении задач с использованием динамического программирования, или к тому, который используется при доказательстве утверждений математической (или структурной) индукции.

3.2.1. Внутреннее состояние является представлением значения абстрактного алгебраического типа данных. Внутреннее представление внутреннего состояния по существу представляет собой (направленное) дерево (или, скорее, направленный ациклический граф) или совокупность узлов; каждый узел содержит некоторые немедленные (обычно целочисленные) значения и несколько указателей на другие (ранее построенные) узлы. При необходимости в начале узла добавляется дополнительный тег конструктора (небольшое целое число), чтобы различать несколько возможностей. Эта структура очень похожа на ту, которая используется для представления значений абстрактных алгебраических типов данных в функциональных языках программирования, таких как Haskell.

3.2.2. Внутреннее состояние является постоянным. Внутреннее состояние является *постоянным* в том смысле, что память, используемая для выделения узлов, которые являются частью внутреннего состояния, никогда не освобождается, пока активен catchain. Кроме того, внутреннее состояние catchain фактически выделяется внутри огромного непрерывного буфера памяти, и новые узлы всегда выделяются в конце используемой части этого буфера путем продвижения указателя. Таким образом, ссылки на другие узлы из узла внутри этого буфера могут быть представлены целочисленным смещением от начала буфера. Каждое внутреннее состояние представлено указателем на его корневой узел внутри этого буфера; этот указатель также может быть представлен целочисленным смещением от начала буфера.

3.2.3. Внутреннее состояние catchain сбрасывается в файл только для добавления. Следствием структуры буфера, используемого для хранения

внутренних состояний catchain, описанного выше, является то, что он обновляется только путем добавления некоторых новых данных в его конце. Это означает, что внутреннее состояние (а точнее буфер, содержащий все необходимые внутренние состояния) catchain может быть сброшено в файл только для добавления и легко восстановлено после перезагрузки. Единственными другими данными, которые необходимо сохранить перед перезагрузкой, является смещение (от начала буфера, т. е. этого файла) текущего состояния catchain. Для этой цели можно использовать простую базу данных ключевого значения.

3.2.4. Обмен данными между различными государствами. Получается, что дерево (а точнее dag), представляющее новое состояние $\sigma_S \cup \{m\} = f(\sigma_S, m)$, разделяет большие поддеревья с предыдущим состоянием σ_S, и, аналогично, $\sigma_S \cup \sigma_T = g(\sigma_S, \sigma_T)$ разделяет большие поддеревья с σ_S и σ_T. Постоянная структура, используемая для представления состояний в BCP, позволяет повторно использовать одни и те же указатели внутри буфера для представления таких общих структур данных вместо их дублирования.

3.2.5. Запоминание узлов. Другим методом, используемым при вычислении новых состояний (т.е. значений функции f), является *запоминание новых узлов*, также заимствованных из функциональных языков программирования. А именно, всякий раз, когда строится новый узел (внутри огромного буфера, содержащего все состояния для определенного catchain), вычисляется его хэш, и простая хэш-таблица используется для поиска последнего узла с тем же хэшем. Если найден узел с этим хэшем, и он имеет такое же содержимое, то вновь построенный узел отбрасывается, и вместо него возвращается ссылка на старый узел с тем же содержимым. С другой стороны, если копия нового узла не найдена, то хэш-таблица обновляется, указатель на конец буфера (выделение) расширяется, а указатель на новый узел возвращается вызывающему объекту.

Таким образом, если различные процессы в конечном итоге выполняют сходные вычисления и имеют сходные состояния, большая часть этих состояний будет разделена, даже если они не связаны напрямую применением функции f, как описано в 3.2.4.

3.2.6. Важность методов оптимизации. Методы оптимизации 3.2.4 и 3.2.5, используемые для совместного использования частей различных

внутренних состояний внутри одного и того же улова, крайне важны для улучшения профиля памяти и производительности BCM в большой группе процессов. Улучшение на несколько порядков в группах $N \approx 100$ процессов. Без этих оптимизаций BCM не подходил бы по назначению (консенсус BFT по новым блокам, генерируемым валидаторами в блокчейне TON).

3.2.7. Сообщение m содержит хэш состояния σ_{Dm+}. Каждое сообщение m содержит хэш (Меркла) (абстрактное представление) соответствующего состояния σ_{Dm+}. Очень грубо, этот хэш вычисляется рекурсивно с использованием дерева узлов представления 3.2.1: все ссылки на узлы внутри узла заменяются (рекурсивно вычисляемыми) хэшами указанных узлов, и вычисляется простой 64-битный хэш результирующей последовательности байтов. Этот хэш также используется для запоминания, как описано в 3.2.5.

Цель этого поля в сообщениях — обеспечить проверку работоспособности вычислений σ_{Dm+}, выполняемых различными процессами (и, возможно, различными реализациями алгоритма обновления состояния): как только σ_{Dm+} вычисляется для вновь доставленного сообщения m, хэш вычисляемого σ_{Dm+} сравнивается со значением, хранящимся в заголовке m. Если эти значения не равны, в журнал ошибок выводится сообщение об ошибке (и никакие дальнейшие действия не предпринимаются программным обеспечением). Эти журналы ошибок могут быть проверены для обнаружения ошибок или несовместимостей между различными версиями BCP.

3.3. Восстановление состояния после перезапуска или сбоев. Улов обычно используется BCP в течение нескольких минут; в течение этого периода программа (программное обеспечение валидатора), работающая по протоколу Catchain, может быть остановлена и перезапущена либо преднамеренно (например, из-за запланированного обновления программного обеспечения), либо непреднамеренно (программа может аварийно завершить работу из-за ошибки в этой или какой-либо другой подсистеме, а затем перезапуститься). Одним из способов справиться с этой ситуацией было бы игнорировать все уловки, не созданные после последнего перезапуска. Однако это приведет к тому, что некоторые валидаторы не будут участвовать в создании каких-либо блоков в течение

нескольких минут (до тех пор, пока не будут созданы следующие экземпляры catchain), что нежелательно. Поэтому протокол восстановления состояния catchain запускается после каждого перезапуска, чтобы валидатор мог продолжать участвовать в том же catchain.

3.3.1. База данных всех доставленных сообщений. С этой целью для каждого активного catchain создается специальная база данных. Эта база данных содержит все известные и доставленные сообщения, проиндексированные по их идентификаторам (хэшам). Для этого достаточно простой базы данных "ключ-значение". Хэш последнего исходящего сообщения $m = m_{i,s}$, сгенерированного текущим процессом i, также хранится в этой базе данных. После перезагрузки все сообщения до m рекурсивно доставляются в правильном порядке (так же, как если бы все эти сообщения были только что получены из сети в произвольном порядке) и обрабатываются протоколом более высокого уровня, пока m, наконец, не будет доставлен, тем самым восстанавливая текущее состояние.

3.3.2. Сброс новых сообщений на диск. Мы уже объясняли в 2.6, что вновь созданные сообщения хранятся в базе данных всех доставленных сообщений (см. 3.3.1) и база данных сбрасывается на диск перед отправкой нового сообщения всем соседям сети. Таким образом, мы можем быть уверены, что сообщение не может быть потеряно, если система аварийно завершает работу и перезагружается, что позволяет избежать создания непроизвольных форков.

3.3.3. Избегание перевычисления состояний σ_{Dm+}. Реализация может использовать файл только для добавления, содержащий все ранее вычисленные состояния, как описано в 3.2.3, чтобы избежать пересчета всех состояний после перезапуска, обменивая дисковое пространство на вычислительную мощность. Однако текущая реализация не использует эту оптимизацию.

3.4. Высокоуровневое описание Протокола консенсуса блока. Теперь мы готовы представить высокоуровневое описание протокола Block Consensus Protocol, используемого валидаторами TON Blockchain для генерации и достижения консенсуса по новым блокам блокчейна. По сути, это трехфазный протокол фиксации, который работает через catchain

(экземпляр протокола Catchain), который используется в качестве «защищенной» системы вещания сообщений в группе процессов.

3.4.1. Создание новых сообщений catchain. Напомним, что низкоуровневый протокол Catchain сам по себе не создает широковещательных сообщений (за единственным исключением — служебные трансляции с доказательствами форка, см. 2.7.5). Вместо этого, когда необходимо создать новое сообщение, протокол более высокого уровня (BCP) предлагается сделать это, вызвав обратный вызов. Кроме того, создание новых сообщений может быть вызвано изменениями текущего виртуального состояния и сигналами тревоги таймера.

3.4.2. Полезная нагрузка сообщений catchain. Таким образом, полезная нагрузка сообщений catchain всегда определяется протоколом более высокого уровня, таким как BCP. Для BCP эта полезная нагрузка состоит из

- Текущее время Unix. Он должен быть неубывающим на последующих сообщениях того же процесса. (Если это ограничение будет нарушено, все процессы, обрабатывающие это сообщение, молчаливо заменят это время Unix максимальным временем Unix, наблюдаемым в предыдущих сообщениях того же отправителя.)
- Несколько (ноль или более) *событий BCP* одного из допустимых типов, перечисленных ниже.

3.4.3. События BCP. Мы только что объяснили, что полезная нагрузка сообщения catchain содержит несколько (возможно, ноль) событий BCP. Теперь перечислим все допустимые типы событий BCP.

- Submit($round,candidate$) — предложите нового кандидата блока
- Approve($round,candidate,signature$) — блок кандидата прошел локальную валидацию
- Reject($round,candidate$) — блок кандидата не прошел локальную проверку
- CommitSign($round,candidate,signature$) — блок кандидата принят и подписан

- Vote($round,candidate$) — голосование за блок кандидата
- VoteFor($round,candidate$) — за этого блок-кандидата необходимо проголосовать в этом туре (даже если текущий процесс имеет другое мнение)
- PreCommit($round,candidate$) — предварительное обязательство перед блок-кандидатом (используется в трехфазной схеме фиксации)

3.4.4. Параметры протокола. Несколько параметров BCP должны быть зафиксированы заранее (в сообщении генезиса catchain, где они инициализируются из значений параметров конфигурации, извлеченных из текущего состояния мастерчейна):

- K — продолжительность одной попытки (в секундах). Это целое количество секунд в текущей реализации; Однако это деталь реализации, а не ограничение протокола
- Y — количество *быстрых* попыток принять кандидата
- C — блок кандидатов, предложенных в ходе одного тура
- Δ_i на $1 \le i \le C$ — отсрочка перед предложением кандидата на блокировку с приоритетом i
- Δ_∞ — отсрочка перед утверждением нулевого кандидата

Возможные значения для этих параметров: $K = 8$, $Y = 3$, $C = 2$, $\Delta_i = 2(i-1)$, $\Delta_\infty = 2C$.

3.4.5. Обзор протокола. BCP состоит из нескольких *раундов*, которые выполняются внутри одного и того же catchain. Более одного раунда могут быть активными в один момент времени, потому что некоторые фазы раунда могут перекрываться с другими фазами других раундов. Поэтому все события BCP содержат явный раунд круглого идентификатора (небольшое целое число, начинающееся с нуля). Каждый раунд завершается либо (коллективно) принятием *блочного кандидата*, предложенного одним из участвующих процессов, либо принятием специального *нулевого кандидата* — фиктивного значения, указывающего, что реальный кандидат

на блок не был принят, например, потому что вообще не были предложены кандидаты на блок. После завершения раунда (с точки зрения участвующего процесса), т.е. как только кандидат на блок собирает подписи CommitSign более $2/3$ всех валидаторов, в этот раунд могут быть добавлены только события CommitSign; процесс автоматически начинает участие в следующем раунде (со следующим идентификатором) и игнорирует все события BCP с различными значениями *раунда*.[6]

Каждый раунд подразделяется на несколько *попыток*. Каждая попытка длится заранее определенный период времени K секунд (BCP использует часы для измерения времени и временных интервалов и предполагает, что часы «хороших» процессов более или менее согласуются друг с другом; поэтому BCP не является асинхронным протоколом BFT). Каждая попытка начинается в Unixtime, точно делящемся на K, и длится в течение K секунд. Попытка идентификатора является Unixtime его начала, деленная на K. Поэтому попытки нумеруются более или менее последовательно 32-битными целыми числами, но не начинаются с нуля. Первые Y попытки раунда быстры; остальные попытки медленные.

3.4.6. Попытка идентификации. Быстрые и медленные попытки. В отличие от раундов, события BCP не имеют параметра, указывающего на попытку, к которой они принадлежат. Вместо этого эта попытка неявно определяется временем Unix, указанным в полезной нагрузке сообщения catchain, содержащего событие BCP (см. 3.4.2). Кроме того, попытки подразделяются на *быстрые* (первые Y попытки раунда, в котором участвует процесс) и *медленные* (последующие попытки того же раунда). Это деление также подразумевается: первое событие BCP, отправленное процессом в раунде, принадлежит определенной попытке, и $попытки\ Y$, начинающиеся с этого, считаются быстрыми в этом процессе.

3.4.7. Производители блоков и кандидаты блоков. В каждом раунде есть C назначенные производители блоков (процессы-члены). (Упорядоченный) список этих производителей блоков вычисляется детерминированным алгоритмом (в простейшем случае процессы i, $i+1$, ..., $i+C-1$ используются в

[6] Это также означает, что каждый процесс неявно определяет Unixtime начала следующего раунда и вычисляет все задержки, например, задержки отправки кандидата в блоки, начиная с этого времени.

i-м раунде, с индексами, взятыми по модулю N, общее количество процессов в catchain) и известен всем участникам без какого-либо дополнительного общения или согласования. Процессы упорядочены в этом списке по снижению приоритета, поэтому первый член списка имеет наивысший приоритет (т.е. если он вовремя предложит кандидата в блок, этот кандидат на блок имеет очень высокие шансы быть принятым протоколом).

Первый производитель блока может предложить кандидата на блок сразу после начала раунда. Другие производители блоков могут предложить кандидатов на блоки только после некоторой задержки Δ_i, где i — индекс производителя в списке назначенных производителей блоков, с $0 = \Delta_1 \leq \Delta_2 \leq \ldots$. По истечении некоторого заданного периода времени Δ_∞ с начала раунда автоматически предполагается специальный *нулевой кандидат* (даже если нет явных событий BCP, указывающих на это). Поэтому в раунде предлагаются максимум кандидаты блока C + 1 (включая нулевого кандидата).

3.4.8. Предложение кандидата на блокировку. Блок-кандидат для TON Blockchain состоит из двух больших «файлов» — блока и собранных данных, а также небольшого заголовка, содержащего описание генерируемого блока (самое главное, полный *идентификатор блока* для кандидата блока, содержащий рабочую цепочку и идентификатор сегмента, порядковый номер блока, хэш файла и его корневой хэш) и хэши sha256 двух больших файлов. Только часть этого небольшого заголовка (включая хэши двух файлов и другие важные данные) используется в качестве *кандидата* в событиях BCP, таких как Submit или CommitSign, для ссылки на конкретного кандидата в блоки. Основная часть данных (самое главное, два больших файла) распространяется в оверлейной сети, связанной с catchain протоколом потокового вещания, реализованным для этой цели через ADNL (см. [3, 5]). Этот механизм массового распространения данных не важен для действительности протокола консенсуса (единственным важным моментом является то, что хэши больших файлов являются частью событий BCP и, следовательно, сообщений catchain, где они подписаны отправителем, и эти хэши проверяются после получения больших файлов любыми участвующими узлами; следовательно, никто не может заменить или повредить эти файлы). Событие BCP Submit($round$,$candidate$) создается в

catchain производителем блока параллельно с распространением block-кандидата, указывая на представление этого конкретного кандидата блока этим производителем блока.

3.4.9. Обработка блоков кандидатов. Как только процесс наблюдает событие Submit BCP в доставленном сообщении catchain, он проверяет действительность этого события (например, его исходный процесс должен быть в списке назначенных производителей, а текущее Unixtime должно быть по крайней мере началом раунда плюс минимальная задержка Δ_i, где i — индекс этого производителя в списке назначенных производителей), и если он действителен, запоминает его в текущем состоянии catchain (ср. 3.1). После этого, когда потоковая трансляция, содержащая связанные с этим блоком-кандидатами (с правильными хэш-значениями), получена (или немедленно, если эти файлы уже присутствуют), процесс вызывает экземпляр валидатора для проверки нового кандидата блока (даже если этот кандидат блока был предложен самим этим процессом!). В зависимости от результата этой проверки создается событие BCP Approve($(round,candidate,signature)$ или Reject($round,candidate$) (и встраивается в новое сообщение catchain). Обратите внимание, что *подпись,* используемая в событиях Утверждения, использует тот же закрытый ключ, который в конечном итоге будет использоваться для подписи принятого блока, но сама подпись отличается от той, которая используется в CommitSign (хэш структуры с другим магическим числом фактически подписан). Поэтому эта промежуточная подпись не может быть использована для подделки принятия этого блока этим конкретным процессом проверки стороннему наблюдателю.

3.4.10. Обзор одного тура. Каждый раунд БКП проходит следующим образом:

- В начале раунда несколько процессов (из заранее определенного списка назначенных производителей) представляют своих кандидатов на блок (с определенными задержками в зависимости от их приоритета производителя) и отражают этот факт с помощью событий Submit (включенных в сообщения catchain).

- Как только процесс получает представленного кандидата блока (т. е. наблюдает за событием Submit и получает все необходимые файлы средствами, внешними по отношению к консенсусному протоколу), он начинает проверку этого кандидата и в конечном итоге создает либо Approve, либо Reject события для этого кандидата на блок.

- Во время каждой быстрой *попытки* (т.е. одной из первых попыток Y) каждый процесс голосует либо за блочного кандидата, набравшего голоса более $2/3$ всех процессов, либо, если таких кандидатов еще нет, за действительных (т.е. одобренных (Approved) более чем $2/3$ всех процессов) заблокировать кандидата с наивысшим приоритетом. Голосование осуществляется путем создания событий Vote (встроенных в новые сообщения catchain).

- Во время каждой *медленной попытки* (т.е. любой попытки, кроме первой Y) каждый процесс голосует либо за кандидата, который был PreCommit ранее (тем же процессом), либо за кандидата, который был предложен VoteFor.

- Если блокирующий кандидат получил голоса более чем от $2/3$ всех процессов во время текущей попытки, и текущий процесс наблюдает за этими голосами (которые собираются в состоянии catchain), создается событие PreCommit, указывающее, что процесс будет голосовать только за этого кандидата в будущем.

- Если кандидат на блок собирает PreCommits из более чем $2/3$ всех процессов внутри попытки, то предполагается, что он принят (группой), и каждый процесс, который наблюдает эти PreCommits, создает событие CommitSign с действительной подписью блока. Эти подписи блока регистрируются в catchain и в конечном итоге собираются для создания «доказательства блока» (содержащего подписи более $2/3$ валидаторов для этого блока). Это доказательство блока является внешним выводом протокола консенсуса (вместе с самим блоком, но без его сопоставленных данных); в конечном итоге он распространяется в оверлейной сети всех полных узлов, которые подписались на новые блоки этого сегмента (или мастерчейна).

- Как только кандидат на блок собирает подписи CommitSign от более чем $2/3$ всех валидаторов, раунд считается завершенным (по крайней мере, с точки зрения процесса, который наблюдает за всеми этими подписями). После этого только CommitSign может быть добавлен к этому раунду этим процессом, и процесс автоматически начинает участвовать в следующем раунде (и игнорирует все события, связанные с другими раундами).

Обратите внимание, что вышеуказанный протокол может привести к подписанию валидатором (в событии CommitSign) кандидата на блок, который был отклонен (Reject) тем же валидатором ранее (это своего рода «подчинение воле большинства»).

3.4.11. Vote и PreCommit создаются детерминированно. Обратите внимание, что каждый процесс может создать не более одного голоса (Vote) и не более одного события PreCommit в каждой попытке. Кроме того, эти события полностью определяются состоянием σ_{Dm} отправителя сообщения catchain m, содержащего такое событие. Таким образом, получатель может обнаружить недействительные события Vote или PreCommit и игнорировать их (тем самым смягчая византийское поведение других участников). С другой стороны, может быть получено сообщение m, которое должно содержать событие Vote или PreCommit в соответствии с соответствующим состоянием σ_{Dm}, но не содержать его. В этом случае текущая реализация автоматически создает недостающие события и протекает так, как если бы m содержал их с самого начала. Однако такие случаи византийского поведения либо исправляются, либо игнорируются (и сообщение выводится в журнал ошибок), но оскорбительные процессы не наказываются иначе (потому что это потребовало бы очень больших доказательств неправомерного поведения для внешних наблюдателей, которые не имеют доступа к внутреннему состоянию catchain).

3.4.12. Множественные Vote(s) и PreCommit(s) одного и того же процесса. Обратите внимание, что процесс обычно игнорирует последующие Vote(s) и PreCommit(s), сгенерированные одним и тем же исходным процессом внутри одной и той же попытки, поэтому обычно процесс может голосовать не более чем за одного кандидата блока. Однако может случиться так, что «хороший» процесс косвенно наблюдает форк, созданный византийским

процессом, с голосами за разных кандидатов блока в разных ветвях этого форка (это может произойти, если «хороший» процесс узнает об этих двух ветвях из двух других «хороших» процессов, которые раньше не видели этого форка). При этом учитываются оба Vote(s) (для разных кандидатов) (добавляются в объединенное состояние текущего процесса). Аналогичная логика применима и к PreCommit(s).

3.4.13. Утверждение или отклонение блокирующих кандидатов. Обратите внимание, что блокирующий кандидат не может быть Approve(d) или Reject(ed) до того, как он был Submit(ted) (т. е. событие Approve, которому не предшествовало соответствующее событие Submit, будет проигнорировано), и что кандидат не может быть утвержден до достижения минимального времени его подачи (время начала раунда плюс приоритетзависимая задержка Δ_i), т.е. любой «хороший» процесс отложит создание его Одобрения (Approve) до этого времени. Кроме того, нельзя утверждать (Approve) более одного кандидата одного и того же производителя в одном раунде (т.е. даже если процесс представляет (Submit) нескольких кандидатов, только один из них — предположительно первый — будет одобрен (Approve) другими «хорошими» процессами; как обычно, это означает, что последующие события утверждения (Approve) будут проигнорированы «хорошими» процессами при получении).

3.4.14. Утверждение кандидата на нулевой блок. Неявный кандидат на нулевой блок также явно утверждается (путем создания события Approve) всеми (хорошими) процессами, как только Δ_∞ задержки с начала раунда.

3.4.15. Выбор блочного кандидата для голосования. Каждый процесс выбирает одного из доступных кандидатов блока (включая неявного нулевого кандидата) и голосует за этого кандидата (путем создания события Vote), применяя следующие правила (в том порядке, в котором они представлены):

- Если текущий процесс создал событие PreCommit для кандидата во время одной из предыдущих попыток, и ни один другой кандидат не набрал голосов более чем из $2/3$ всех процессов с тех пор (т. Е. Внутри одной из последующих попыток, включая текущую до сих пор; мы

говорим, что событие PreCommit все еще *активно* в этом случае), затем текущий процесс снова голосует за этого кандидата.

- Если текущая попытка быстрая (т.е. одна из первых Y попыток раунда с точки зрения текущего процесса), и кандидат набрал голоса из более чем $2/3$ всех процессов во время текущей или одной из предыдущих попыток, текущий процесс голосует за этого кандидата. В случае ничьей выбирается кандидат из последней из всех таких попыток.

- Если текущая попытка быстрая, и предыдущие правила не применяются, то процесс голосует за кандидата с наивысшим приоритетом среди всех *правомочных кандидатов*, то есть кандидатов, которые собрали Одобрения (Approve(s)) (наблюдаемые текущим процессом) из более чем $2/3$ всех процессов.

- Если текущая попытка медленная, то процесс голосует только после того, как он получит действительное событие VoteFor в той же попытке. Если применяется первое правило, процесс голосует в соответствии с ним (т.е. за ранее предварительно принятого (PreCommit(ed)) кандидата). В противном случае он голосует за кандидата от блока, который указан в событии VoteFor. При наличии нескольких таких допустимых событий (во время текущей попытки) выбирается кандидат с наименьшим хешем (это может произойти в редких ситуациях, связанных с различными событиями VoteFor, созданными в разных ветвях форка, см. 3.4.12).

Считается, что «нулевой кандидат» имеет наименьший приоритет. Он также требует явного одобрения (Approve) перед голосованием (за исключением первых двух правил).

3.4.16. Создание событий VoteFor во время медленных попыток. Событие VoteFor создается в начале медленной попытки координатора — процесса с индексом *попытки* $mod\ N$ в упорядоченном списке всех процессов, участвующих в catchain (как обычно, это означает, что VoteFor, созданный другим процессом, будет проигнорирован всеми «хорошими» процессами). Это событие VoteFor относится к одному из кандидатов блока (включая нулевого кандидата), который собрал Одобрения (Approve(s)) из более чем $2/3$ всех процессов, обычно случайным образом выбранных среди всех

таких кандидатов. По сути, это предложение проголосовать за этого кандидата блока, направленное на все другие процессы, которые не имеют активного PreCommit.

3.5. Срок действия BCP. Теперь мы представляем эскиз доказательства действительности протокола TON Block Consensus Protocol (BCP), описанный выше в 3.4, предполагая, что менее трети всех процессов демонстрируют византийское (произвольно злонамеренное, возможно, нарушающее протокол) поведение, как это принято для византийских отказоустойчивых протоколов. В этом подразделе мы рассмотрим только один раунд BCP, разделенный на несколько попыток.

3.5.1. Фундаментальное предположение. Подчеркнем еще раз, что мы предполагаем, что *менее трети всех процессов являются византийскими*. Все остальные процессы считаются хорошими, т.е. они следуют протоколу.

3.5.2. Взвешенный BCP. Рассуждения в этом подразделе справедливы и для *взвешенного варианта BCP*. В этом варианте каждому процессу $i \in I$ предварительно присваивается положительный вес $w_i > 0$ (зафиксированный в сообщении генезиса catchain), а утверждения о «более чем $2/3$ всех процессов» и «менее одной трети всех процессов» понимаются как «более $2/3$ всех процессов по весу», т. е. «подмножество $J \subset I$ процессов с общим весом $\sum_{j\in J} w_j > \frac{2}{3}\sum_{i\in I} w_i$", и аналогично для второго имущества. В частности наше «фундаментальное предположение» 3.5.1 следует понимать в том смысле, что «общий вес всех византийских процессов составляет менее одной трети от общего веса всех процессов».

3.5.3. Полезные инварианты. Мы собираем здесь некоторые полезные инварианты, которым подчиняются все события BCP во время одного раунда BCP (внутри catchain). Эти инварианты реализуются двумя способами. Во-первых, любой «хороший» (невизантийский) процесс не создаст событий, нарушающих эти инварианты. Во-вторых, даже если «плохой» процесс создает событие, нарушающее эти инварианты, все «хорошие» процессы обнаружат это, когда сообщение catchain, содержащее это событие, доставляется в BCP и игнорирует такие события. Некоторые возможные проблемы, связанные с вилками (см. пункт 3.4.12), сохраняются даже после принятия этих мер предосторожности; мы

указываем, как эти вопросы решаются отдельно, и игнорируем их в этом списке. Так:

- В каждом процессе существует не более одного события Submit (в пределах одного раунда BCP).
- В каждом процессе, связанном с одним кандидатом, существует не более одного события Approve или Reject (точнее, даже если есть несколько кандидатов, созданных одним и тем же назначенным производителем блока, только один из них может быть одобрен (Approve(d) другим процессом).[7] Это достигается путем требования ко всем «хорошим» процессам игнорировать (т.е. не создавать Approve или Reject) всех кандидатов, предложенных одним и тем же производителем, но самым первым, о котором они узнали.
- Во время каждой попытки каждый процесс проводит не более одного голосования (Vote) и не более одного события PreCommit.
- Во время каждой (медленной) попытки проводится не более одного события VoteFor.
- В каждом процессе существует не более одного события CommitSign.
- Во время медленной попытки каждый процесс голосует либо за своего ранее предварительно назначенного (Precommit(ted))

[7] На самом деле отказы появляются только в этом ограничении, и больше ни на что не влияют. Поэтому любой процесс может воздерживаться от отправки Rejects без нарушения протокола, а события Reject могли быть полностью удалены из протокола. Вместо этого текущая реализация протокола по-прежнему генерирует Rejects, но ничего не проверяет на их получение и не запоминает их в состоянии catchain. В журнал ошибок выводится только сообщение, а кандидат-нарушитель сохраняется в специальном каталоге для дальнейшего изучения, поскольку отклонения обычно указывают либо на наличие византийского противника, либо на ошибку в программном обеспечении коллатора (генерация блока) или валидатора (проверка блока) либо на узле, предложившем блок, либо на узле, создавшем событие Reject.

кандидата, либо за кандидата, указанного в VoteFor в случае этой попытки.

Можно было бы несколько улучшить приведенные выше утверждения, добавив слово «действительный», где это уместно (например, существует не более одного *действительного* события Submit ...).

3.5.4. Больше инвариантов.

- Существует не более одного подходящего кандидата (т.е. кандидата, который получил одобрение от более чем $2/3$ всех процессов) от каждого назначенного производителя, и нет подходящих кандидатов от других производителей.
- В общей сложности есть не более $C + 1$ подходящих кандидатов (не более C кандидатов от назначенных *производителей* C, плюс нулевой кандидат).
- Кандидат может быть принят только в том случае, если он собрал более $2/3$ PreCommit(s) за одну и ту же попытку (точнее, кандидат принимается только в том случае, если есть события PreCommit, созданные более чем $2/3$ всех процессов для этого кандидата и принадлежащие к одной и той же попытке).
- Кандидат может быть проголосован (Vote(d)), предварительно принят (PreCommit(ted)) или упомянут в VoteFor только в том случае, если он является *правомочным кандидатом*, что означает, что он ранее собрал Одобрения (Approve(s)) от более чем $2/3$ всех валидаторов (т. е. действительное событие голосования (Vote) может быть создано для кандидата только в том случае, если события одобрения (Approve) для этого кандидата были ранее созданы более чем $2/3$ всех процессов и зарегистрированных в сообщениях catchain, наблюдаемых из сообщения, содержащего событие Vote, и аналогично для событий PreCommit и VoteFor).

3.5.5. Принимается не более одного блока кандидата. Теперь мы утверждаем, что *максимум один блок кандидатов может быть принят (в*

раунде BCP). Действительно, кандидат может быть принят только в том случае, если он собирает PreCommit(s) из более чем $2/3$ всех процессов внутри одной и той же попытки. Таким образом, два разных кандидата не могут достичь этого во время одной и той же попытки (в противном случае более одной трети всех валидаторов должны были создать PreCommit(s) для двух разных кандидатов внутри попытки, тем самым нарушая вышеуказанные инварианты; но мы предположили, что менее одной трети всех валидаторов демонстрируют византийское поведение). Теперь предположим, что два разных кандидата c_1 и c_2 собрали PreCommit(s) из более чем $2/3$ всех процессов в двух разных попытках a_1 и a_2. Можно предположить, что $a_1 < a_2$. Согласно первому правилу 3.4.15, каждый процесс, создавший PreCommit для c_1 во время попытки a_1, должен продолжать голосовать за c_1 во всех последующих попытках $a' > a_1$ или, по крайней мере, не может голосовать за любого другого кандидата, если только другой кандидат c' не наберет голосов (Vote(s)) более $2/3$ всех процессов во время последующей попытки (и этот инвариант применяется, даже если некоторые процессы пытаются не создавать эти новые события Vote для c_1, см. 3.4.11). Поэтому, если $c_2 \neq c_1$ собрал необходимое количество PreCommit(s) во время попытки $a_2 > a_1$, есть по крайней мере одна попытка a', $a_1 < a' \leq a_2$, так что некоторые $c' \neq c_1$ (не обязательно равны c_2) собрали голоса (Vote(s)) более $2/3$ всех процессов во время попытки a'. Зафиксируем наименьшую *такую* a', и соответствующую $c' \neq c_1$, которая собрала много голосов во время попытки a'. Более $2/3$ всех валидаторов проголосовали за c' во время попытки a', и более $2/3$ всех валидаторов предварительно обязались (PreCommit(ted)) за c_1 во время попытки a_1, и по минимальности a' не было попытки a'' с $a_1 < a'' < a'$, такой, что кандидат, отличный от c_1, набрал более $2/3$ всех голосов во время попытки a''. Таким образом, все валидаторы, которые предварительно обязались (PreCommit(ted)) за c_1, могли голосовать только за c_1 во время попытки a', и в то же время мы предполагали, что c' собрал голоса более чем от $2/3$ всех валидаторов во время одной и той же попытки a'. Это означает, что более $1/3$ всех валидаторов каким-то образом проголосовали как за c_1, так и *за* c' во время этой попытки (или проголосовали за c', в то время как они могли проголосовать только за c_1), то есть более $1/3$ всех валидаторов проявили византийское поведение. Это невозможно по нашему фундаментальному предположению 3.5.1.

3.5.6. Максимум один блок-кандидат может быть предварительно зафиксирован (PreCommit(ted)) в течение одной попытки. Обратите внимание, что все допустимые события PreCommit (если таковые имеются), созданные внутри одной и той же попытки, должны ссылаться на одного и того же кандидата в блок, по тем же рассуждениям, что и в первой части 3.5.5: поскольку действительное событие PreCommit для кандидата *c* может быть создано только после того, как голоса из более чем $2/3$ всех процессов наблюдаются для этого кандидата внутри той же попытки (и недопустимые PreCommits игнорируются всеми хорошими процессами), существование действительных событий PreCommit для разных кандидатов c_1 и c_2 внутри одной и той же попытки означало бы, что более трети всех процессов голосовали как за c_1, так и *за* c_2 внутри этой попытки, т. е. они демонстрировали византийское поведение. Это невозможно с учетом нашего фундаментального предположения 3.5.1.

3.5.7. Предыдущая PreCommit деактивируется наблюдением более новой. Мы утверждаем, что *всякий раз, когда процесс с активным* PreCommit *наблюдает действительный* PreCommit, *созданный любым процессом в последующей попытке для другого кандидата, его ранее* активный PreCommit *деактивируется*. Напомним, что мы говорим, что процесс имеет активный PreCommit, если он создал PreCommit для определенного кандидата c во время определенной попытки a, не создал PreCommit во время любых попыток $a' > a$ и не наблюдал голосов более $2/3$ всех валидаторов для любого кандидата $\neq c$ во время любых попыток $a' > a$. Любой процесс имеет не более одного активного PreCommit, и если он у него есть, он должен голосовать (Vote) только за предварительно преданного кандидата.

Теперь мы видим, что если процесс с активным PreCommit для кандидата c с момента попытки a соблюдает действительный PreCommit (обычно другим процессом) для кандидата c', созданный во время некоторой последующей попытки $a' > a$, то первый процесс должен также соблюдать все зависимости сообщения, содержащего более новый PreCommit; эти зависимости обязательно включают действительные голоса от более чем $2/3$ всех валидаторов для одного и того же кандидата $c' \neq c$, созданных во время одной и той же попытки $a' > a$ (потому что в противном случае более новый PreCommit не был бы действителен и был бы

проигнорирован другим процессом); по определению, наблюдение за всеми этими голосами (Vote(s) деактивирует исходный PreCommit.

3.5.8. Допущения для доказательства конвергенции протокола. Теперь мы собираемся доказать, что протокол, описанный выше, *сходится* (т.е. завершается после принятия кандидата на блок) с вероятностью один при некоторых предположениях, которые по существу говорят нам, что существует достаточно «хороших» процессов (т.е. процессов, которые усердно следуют протоколу и не вводят произвольных задержек перед отправкой своих новых сообщений), и что эти хорошие процессы пользуются хорошей сетевой связью, по крайней мере, время от времени. Точнее, наши предположения таковы:

- Существует подмножество $I^+ \subset I$, состоящее из «хороших» процессов и содержащее более $2/3$ всех процессов.

- Все процессы из I^+ имеют хорошо синхронизированные тактовые частоты (отличающиеся не более *чем* τ, где τ — привязка к сетевой задержке, описанная ниже).

- Если попыток бесконечно много, то бесконечно много попыток являются «хорошими» в отношении сетевой связи между процессами из I^+, что означает, что все сообщения, созданные процессом из I^+ во время этой попытки или ранее, доставляются в любой другой процесс из I^+ в пределах не более $\tau > 0$ секунд после создания с вероятностью не менее $q > 0$, где $\tau > 0$ и $0 < q < 1$ — некоторые фиксированные параметры, такие как $5\tau < K$, где K — продолжительность одной попытки.

- Кроме того, если протокол работает для бесконечно многих попыток, то любая арифметическая прогрессия попыток содержит бесконечно много «хороших» попыток в описанном выше смысле.

- Процесс из I^+ создает VoteFor во время медленной попытки после некоторой фиксированной или случайной задержки после начала медленной попытки, таким образом, что эта задержка относится к интервалу $(\tau, K-3\tau)$ с вероятностью не менее q', где $q' > 0$ является фиксированным параметром.

- Процесс от I^+, когда наступает его очередь быть координатором медленной попытки, выбирает кандидата для VoteFor равномерно среди всех подходящих кандидатов (т. е. тех кандидатов, которые собрали Одобрения (Approve) от более чем $2/3$ всех валидаторов).

3.5.9. Протокол прекращается в соответствии с этими предположениями. Теперь мы утверждаем, что *(каждый раунд) протокола BCP, как описано выше, завершается с вероятностью один в соответствии с предположениями, перечисленными в* 3.5.8. Доказательство исходит из следующего.

- Предположим, что протокол не сходится. Затем он продолжает работать вечно. Мы проигнорируем первые несколько попыток, и рассмотрим только попытки a_0, $a_0 + 1$, $a_0 + 2$, ... начиная с какого-то a_0, чтобы быть выбранным позже.

- Поскольку все процессы из I^+ продолжают участвовать в протоколе, они создадут хотя бы одно сообщение не намного позже начала раунда (которое может восприниматься немного по-разному каждым процессом). Например, они создадут Approve для нулевого кандидата не позднее, чем через Δ_∞ секунд с начала раунда. Поэтому после этого они будут считать все попытки медленными максимум через несколько секунд KY. Выбрав a_0 соответствующим образом, мы можем предположить, что все рассматриваемые нами попытки медленны с точки зрения всех процессов из I^+.

- После «хорошей» попытки $a \geq a_0$ все процессы из I^+ увидят Approve(s) для нулевого кандидата, созданного всеми другими процессами из I^+, и будут считать нулевого кандидата правомочным отныне. Поскольку «хороших» попыток бесконечно много, это рано или поздно произойдет с вероятностью один. Таким образом, мы можем предположить (увеличивая a_0 , если это необходимо), что есть по крайней мере один подходящий кандидат с точки зрения всех процессов из I^+, а именно нулевой кандидат.

- Кроме того, будет бесконечно много попыток $a \geq a_0$, которые воспринимаются медленными всеми процессами из I^+, которые

имеют координатора от I^+, и которые являются «хорошими» (в отношении сетевого подключения), как определено в 3.5.8. Назовем такие попытки «очень хорошими».

- Рассмотрим одну «очень хорошую» медленную попытку a. С вероятностью $q' > 0$ его координатор (который принадлежит I^+) будет ждать $\tau' \in (\tau, K - 3\tau)$ секунд перед созданием своего события VoteFor. Рассмотрим самое последнее событие PreCommit, созданное любым процессом из I^+; предположим, что оно было создано во время попытки $a' < a$ для некоторого кандидата c'. С вероятностью $qq' > 0$ сообщение catchain, несущее этот PreCommit, уже будет доставлено координатору во время генерации его события VoteFor. В этом случае сообщение catchain, несущее это событие VoteFor, будет зависеть от этого события PreCommit(c'), и все «хорошие» процессы, которые наблюдают за этим VoteFor, также будут соблюдать его зависимости, включая этот PreCommit(c'). Мы видим, что *с вероятностью не менее qq', все процессы из I^+, которые получают* событие VoteFor *в течение «очень хорошей» медленной попытки получить также самый последний* PreCommit *(если таковой имеется).*

- Затем рассмотрите любой процесс от I^+ , который получает этот VoteFor, для случайно выбранного приемлемого кандидата c, и предположите, что уже есть некоторые PreCommits, и что предыдущее утверждение выполняется. Поскольку есть не более $C+1$ подходящих кандидатов (ср. 3.5.4), с вероятностью не менее $1/(C+1) > 0$ мы получим $c = c'$, где c' является самым последним предварительно принятым кандидатом (есть максимум один такой кандидат на 3.5.6). В этом случае все процессы из I^+ будут голосовать за $c = c'$ во время этой попытки сразу после получения этого VoteFor (который будет доставлен любому процессу $j \in I^+$ менее $K - 2\tau$ секунд после начала попытки с вероятностью qq'). Действительно, если процесс j из I^+ не имел активного PreCommit, он будет голосовать за значение, указанное в VoteFor, которое равно c. Если j имел активный PreCommit, и он является как можно более недавним, т.е. также созданным во время попытки a', то это должен был быть PreCommit для того же значения $c' = c$ (потому что мы знаем по крайней мере об

одном допустимом PreCommit для c' во время попытки a', а все остальные допустимые PreCommits во время попытки a' должны быть для того же c' по пункту 3.5.6). Наконец, если $y\ j$ был активный PreCommit от попытки $< a0$, то он станет неактивным, как только VoteFor со всеми его зависимостями (включая более новый PreCommit(c') будет доставлен в этот процесс j (см. 3.5.7), и процесс снова проголосует за значение c, указанное в VoteFor. Поэтому все процессы из I^+ будут голосовать за одно и то же $c = c'$ во время этой попытки, менее К -2τ секунд после начала попытки (с некоторой вероятностью, ограниченной от нуля).

- Если Precommit(s) еще нет, то приведенные выше рассуждения еще больше упрощаются: все процессы от I^+, которые получают этот VoteFor, немедленно проголосуют за кандидата c, предложенного этим VoteFor.

- В обоих случаях все процессы из I^+ создадут Vote за одного и того же кандидата c менее $K-2\tau$ секунд от начала попытки, и это произойдет с положительной вероятностью, ограниченной от нуля.

- Наконец, все процессы из I^+ получат эти Голоса за c от всех процессов из I^+, опять же меньше $(K - 2\tau) + \tau = K - \tau$ секунд после начала этой попытки, т.е. все еще во время той же попытки (даже с учетом несовершенной синхронизации часов между процессами из I^+). Это означает, что все они создадут действительный PreCommit для c, то есть протокол примет c во время этой попытки с вероятностью, ограниченной от нуля.

- Поскольку существует бесконечно много «очень хороших» попыток, и вероятность успешного завершения во время каждой такой попытки составляет $\geq p > 0$ для некоторого фиксированного значения p, протокол будет успешно завершаться с вероятностью один.

Ссылки

[1] K. Birman, Reliable Distributed Systems: Technologies, Web Services and Applications, Springer, 2005.

[2] M. Castro, B. Liskov, et al., Practical byzantine fault tolerance, Proceedings of the Third Symposium on Operating Systems Design and Implementation (1999), p. 173–186, available at http://pmg.csail.mit. edu/papers/osdi99.pdf.

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

[4] N. Durov, Telegram Open Network Blockchain, 2018.

[5] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Transactions on Programming Languages and Systems, 4/3 (1982), p. 382–401.

[6] A. Miller, Yu Xia, et al., The honey badger of BFT protocols, Cryptology e-print archive 2016/99, https://eprint.iacr.org/2016/ 199.pdf, 2016.

[7] M. van Steen, A. Tanenbaum, Distributed Systems, 3rd ed., 2017.