

# Catchain Consensus: An Outline

Nikolai Durov  
TL: Dr Awesome Doge

August 12, 2023

## **Abstract**

本文的目的是提供 Catchain 共識協議的概述，它是一種特別為 TON 區塊鏈中的塊生成和驗證而製定的拜占庭容錯（BFT）協議 [3]。該協議可能也可以用於 PoS 區塊鏈中的其他目的，但目前的實現僅使用了某些僅對這個特定問題有效的優化。

---

## Contents

|          |                                 |           |
|----------|---------------------------------|-----------|
| <b>1</b> | <b>Overview</b>                 | <b>3</b>  |
| <b>2</b> | <b>Catchain 協議</b>              | <b>5</b>  |
| <b>3</b> | <b>Block Consensus Protocol</b> | <b>14</b> |

# 1 Overview

Catchain 共識協議建立在 TON Network ([3]) 的疊加網路構建協議和疊加網路廣播協議之上。Catchain 共識協議本身可以分解為兩個獨立的協議，一個更低層次和通用的協議（*Catchain* 協議<sup>1</sup>），另一個是高層次的塊共識協議（*BCP*），它使用了 *Catchain* 協議。在 TON 協議棧中，更高的層次被塊生成和驗證層佔據；然而，所有這些都基本上在一個（邏輯）機器上本地執行，而新生成的塊達成共識的問題被委派給 *Catchain* 協議層。

以下是 TON 區塊生成和分佈所使用的協議堆棧的近似圖表，顯示了 *Catchain Consensus* 協議（或其兩個組件協議）的正確位置：

- 頂層：區塊生成和區塊驗證軟體，邏輯上運行在獨立的邏輯機器上，所有的輸入和輸出都由較低級的協議處理。這個軟體的工作是生成區塊鏈（shardchain 或 TON 區塊鏈的主鏈；有關 shardchain 和主鏈的討論，請參見 [3]），或檢查由他人生成的區塊的有效性。
- (*TON*) 區塊共識協議：在主鏈或 shardchain 的當前驗證者組中實現（拜占庭容錯）共識，以接受下一個區塊。這一級別利用區塊生成和驗證軟體的（抽象接口），並在較低級的 *Catchain* 協議之上構建。有關此協議的詳細信息，請參見第 3 節。
- *Catchain* 協議：在疊加網絡中提供安全的持久性廣播（例如，特定 shardchain 的驗證者任務組或用於在此 shardchain 或主鏈中生成、驗證和傳播新區塊的主鏈），並檢測一些參與者的「欺詐」（協議違規）嘗試。有關此協議的詳細信息，請參見第 2 節。
- (*TON Network*) 疊加廣播協議：是 TON Network 中疊加網絡的一個簡單盡力而為的廣播協議，如 [3] 所述。只需將接收到的廣播訊息廣播到在同一個疊加網絡中且尚未收到這些訊息副本的所有鄰居，並盡最小的努力在短時間內保留未傳遞的廣播訊息的副本。
- (*TON Network*) 疊加協議：在 ADNL 協議網絡中創建疊加網絡（參見 [3]），管理這些疊加網絡的鄰居列表。疊加網絡的每個參與者在同一個疊加網絡中跟踪多個鄰居，並保持專門的 ADNL 連接（稱為「通道」），以便傳入訊息可以以最小的開銷高效地廣播到所有鄰居。

---

<sup>1</sup>在研究和開發階段的初期，該協議的原始名稱是 *catch-chain* 或 *catchchain*，因為它本質上是一個專門用於捕獲共識協議中所有重要事件的特殊鏈；在多次說和寫這個名字之後，它逐漸縮短為“catchain”。

- 抽象數據報文網絡層 (ADNL) 協議：是 TON Network 的基本協議，僅使用 256 位抽象 (ADNL) 地址識別僅由其加密金鑰 (或其哈希) 識別的網絡節點之間的封包 (數據報文)。

本文旨在僅描述此套件中的第二個和第三個協議，即 (TON) 區塊共識協議和 (TON) Catchain 協議。

我們在此指出，本文的作者雖然提供了此協議應該如何設計的一般指南 (基於「讓我們創建一個 BFT-hardened 群組廣播訊息系統，並在此系統之上運行適當適應的簡單的兩階段或三階段提交協議」的思路)，並參與了協議的開發和實現過程中的多次討論，但絕不是此協議以及其當前實現的唯一設計者。這是多個人的工作。

對於合併的 Catchain Consensus 協議的效率，我們有幾句話要說。首先，它是一個真正的拜占庭容錯 (Byzantine Fault Tolerant, BFT) 協議，即使一些參與者 (驗證者) 表現出任意惡意行為，只要這些惡意參與者少於驗證者總數的三分之一，它也能夠最終達成對於區塊鏈上下一個有效區塊的共識。已經廣泛認識到，如果參與者中至少有三分之一是惡意的，那麼實現 BFT 共識是不可能的 (參見 [5])，因此，在這方面，Catchain Consensus 協議在理論上是最好的。

其次，當 Catchain Consensus 協議首次在全球分佈的多達 300 個節點上實現並進行測試時 (在 2018 年 12 月)，即使一些節點未能參與或表現不正確的行為，它也能夠在 300 個節點上在 6 秒內達成新區塊的共識，在 100 個節點上則在 4-5 秒內完成 (在 10 個節點上則為 3 秒)。<sup>2</sup> 由於 TON 區塊鏈任務組不會超過 100 個驗證者 (即使總共有一千或一萬個驗證者在運行，只有 100 個擁有最大股份的驗證者將生成新的主鏈區塊，其他驗證者只會參與新的分片鏈區塊的生成，每個分片鏈區塊由 10-30 個驗證者生成和驗證；當然，這裡給出的所有數字都是配置參數 (參見 [3] 和 [4])，如果需要，可以通過驗證者的共識投票進行調整)，這意味著 TON 區塊鏈能夠按照最初的計劃每 4-5 秒生成新區塊。這個承諾在幾個月後 (2019 年 3 月) 推出的 TON 區塊鏈測試網絡中得到了進一步的驗證和履行。因此，我們可以看出，Catchain Consensus 協議是實用 BFT 協議不斷增長家族中的新成員 (參見 [2])，即使它基於略有不同的原則。

---

<sup>2</sup>當惡意、未參與或非常緩慢的驗證者的比例增長到三分之一時，協議會出現優雅的退化，區塊共識時間增長非常緩慢，最多增長半秒，直到幾乎達到三分之一的臨界值。

## 2 Catchain 協議

在概述部分（參見1）中，我們已經解釋了 TON 區塊鏈所使用的 BFT 共識協議，以實現對新區塊鏈塊的共識。該共識協議由兩個協議組成。在此，我們提供了對這兩個協議中的低層協議——Catchain 協議的簡要描述。該協議除了用於 BFT 共識協議外，還有可能用於其他用途。Catchain 協議的源代碼位於源代碼目錄下的 `catchain` 子目錄中。

**2.1. Prerequisites for running the Catchain protocol.** 運行 Catchain 協議（一個實例）的主要前提條件是有參與（或被允許參與）該協議實例的所有節點的有序列表。該列表包含所有參與節點的公鑰和 ADNL 地址。在創建 Catchain 協議實例時，需要從外部提供此列表。

**2.2. Nodes participating in the block consensus protocol.** 對於為 TON 區塊鏈的某一個區塊鏈（即主區塊鏈或活動分片鏈）創建新區塊的特定任務，會創建一個由多個驗證者組成的特殊任務組。此任務組成員的列表既用於在 ADNL 內創建私有疊加網絡（這意味著只有在其創建期間明確列出的節點才能加入此疊加網絡），也用於運行相應的 Catchain 協議實例。

成員列表的構建是整個協議堆棧（即區塊創建和驗證軟件）的高層級責任，因此不是本文的主題（[4] 是更適當的參考文獻）。目前只需要知道，此列表是當前（最近）主區塊鏈狀態（尤其是配置參數的當前值，例如所有選舉用於創建新區塊的驗證者的活動列表以及其各自的權重）的一個確定性函數。由於列表是確定性計算的，所有驗證者計算出相同的列表，特別是每個驗證者知道自己參與的任務組（即 Catchain 協議實例），無需進一步進行網絡通信或協商。<sup>3</sup>

**2.2.1. Catchains are created in advance.** 實際上，不僅計算了上述列表的當前值，還計算了它們的立即後續（未來）值，以便提前創建 Catchain。這樣，當驗證者任務組的新實例需要創建第一個區塊時，Catchain 已經準備就緒。

**2.3. The genesis block and the identifier of a catchain.** 一個 *catchain*（即 Catchain 協議的一個實例）的特徵是它的創世塊 或創世消息。它是一個簡單的數據結構，包含一些魔數、catchain 的目的（例如要生成區塊的分片鏈的標識符，以及所謂的 *catchain* 序列號，也是從主區塊鏈配置中獲得，用於區分生成“相同”的分片鏈的後續 catchain 實例，但可能具有不同的

---

<sup>3</sup>如果某些驗證者的主區塊鏈狀態已過時，它們可能無法計算出正確的任務組列表，也無法參與相應的 Catchain；在這方面，它們被視為惡意或故障，只要不到三分之一的所有驗證者以此方式失敗，就不會影響整個 BFT 協議的有效性。

參與驗證者)，最重要的是所有參與節點的列表（其 ADNL 地址和 Ed25519 公鑰，如 2.1 所述）。Catchain 協議本身僅使用此列表和整個數據結構的 SHA256 哈希值；該哈希值用作 catchain 的內部標識符，即該 Catchain 協議的特定實例的標識符。

**2.3.1. Distribution of the genesis block.** 請注意，創世塊不會在參與節點之間分發；相反，每個參與節點都會獨立計算，如 2.2 所述。由於創世塊的哈希用作 catchain 的標識符（即 Catchain 協議的特定實例的標識符；參見 2.3），如果節點（意外或故意地）計算出不同的創世塊，它將被有效地鎖定，無法參與“正確”的協議實例。

**2.3.2. List of nodes participating in a catchain.** 請注意，參與 Catchain 的節點（有序）列表在創世塊中固定，因此它對所有參與者都是已知的，並且可以通過創世塊的哈希（即 catchain 標識符）明確確定，前提是 SHA256 沒有（已知的）碰撞。因此，在下面討論一個特定的 catchain 時，我們固定參與節點的數量  $N$ ，並假設節點從 1 到  $N$  編號（可以使用此範圍內的索引在參與者列表中查找它們的真實身份）。所有參與者的集合將用  $I$  表示；我們假設  $I = 1 \dots N$ 。

**2.4. Messages in a catchain. Catchain as a process group.** 一個觀點是，Catchain 是由  $N$  個已知且固定的「（通信）進程」（或前面所述的「節點」）組成的「（分布式）進程組」，這些進程生成「廣播消息」，最終廣播給進程組中的所有成員。所有進程的集合由  $I$  表示；通常假設  $I = 1 \dots N$ 。每個進程生成的廣播從一開始編號，因此進程  $i$  的第  $n$  個廣播會收到「序列號」或「高度」 $n$ ；每個廣播都應該由發起進程的身份或索引  $i$  和其高度  $n$  唯一確定，因此我們可以將  $(i, n)$  視為進程組內廣播消息的自然標識符。<sup>4</sup> 由同一進程  $i$  生成的廣播消息預計以完全相同的順序傳送到每個其他進程，即按照它們的高度遞增的順序。在這方面，Catchain 和 [1] 或 [7] 中的進程組非常相似。其主要區別在於，Catchain 是進程組的「強化」版本，能夠容忍某些參與者的可能的拜占庭（任意惡意）行為。

**2.4.1. Dependence relation on messages.** 可以在進程組中廣播的所有消息上引入一個「依賴關係」。該關係必須是一個嚴格的偏序  $\prec$ ，滿足  $m_{i,k} \prec m_{i,k+1}$  的性質，其中  $m_{i,k}$  表示具有索引  $i$  的進程成員廣播的第  $k$  條消息。 $m \prec m'$  的含義是  $m'$  依賴於  $m$ ，因此（廣播）消息  $m'$  只能在  $m$  被處理（由進程組的成員處理）之後才能被處理。例如，如果消息  $m'$  表示進程組成員對另一個消息  $m$  的反應，則將  $m \prec m'$  設置為自然的。如果進程組的一個成員在其依賴關係中的消息  $m \prec m'$  都被處理（或已被高級協議

---

<sup>4</sup>在 Catchain 的拜占庭環境中，這在某些情況下不一定是真的。

「交付」) 之前收到了消息  $m'$ , 則其處理 (或「交付」) 被延遲, 直到所有依賴項都被交付。

我們將依賴關係定義為嚴格偏序, 因此它必須是可傳遞性的 ( $m'' \prec m'$  和  $m' \prec m$  意味著  $m'' \prec m$ ), 反對稱的 (對於任何兩條消息  $m$  和  $m'$ ,  $m' \prec m$  和  $m \prec m'$  最多只能有一條成立) 和反自反的 ( $m \prec m$  從未成立)。如果我們有一個更小的「基本依賴關係」集合  $m' \rightarrow m$ , 則可以構造其可傳遞性閉包  $\rightarrow^+$ , 並將  $\prec := \rightarrow^+$ 。唯一的其他要求是每個發送者的廣播都依賴於相同發送者的所有先前廣播。嚴格來說, 不一定需要假設這一點; 然而, 這種假設非常自然且大大簡化了進程組內部消息系統的設計, 因此 Catchain 協議採用了這個假設。

**2.4.2. Dependence set or cone of a message.** 假設  $m$  是如上所述的進程組中的 (廣播) 消息。我們稱集合  $D_m := \{m' : m' \prec m\}$  為消息  $m$  的依賴集合 / 或依賴錐 /。換句話說,  $D_m$  是在所有消息的有限偏序集合中由  $m$  生成的主理想集合 /。它正是在  $m$  交付之前必須交付的所有消息的集合。

**2.4.3. Extended dependence cone of a message.** 我們還通過  $D_m^+ := D_m \cup m$  定義  $m$  的擴展依賴錐。

**2.4.4. Cones, or ideals with respect to  $\prec$ .** 更一般地, 我們稱消息的子集  $D$  為錐, 如果它對於依賴關係  $\prec$  是一個理想, 即如果  $m \in D$  和  $m' \prec m$  則意味著  $m' \in D$ 。當然, 任何消息  $m$  的依賴錐  $D_m$  和擴展依賴錐  $D_m^+$  都是錐 (因為在偏序集合中的任何主理想都是一個理想)。

**2.4.5. Identification of cones with the aid of vector time.** 回想一下, 我們已經假設任何消息都依賴於同一發送者的所有先前消息, 即對於任何  $i \in I$  和任何  $s > 0$ , 滿足  $m_{i,s+1}$  存在, 有  $m_{i,s} \prec m_{i,s+1}$ 。這意味著任何錐  $D$  都由  $N$  個由  $i \in I$  索引的值  $\text{VT}(D)_i$  完全描述:

$$\text{VT}(D)_i := \sup\{s \in \mathbb{N} : m_{i,s} \in D\} = \inf\{s \in \mathbb{N}_0 : m_{i,s+1} \notin D\} \quad (1)$$

(如果沒有消息  $m_{i,s}$  在  $D$  中, 我們設置  $\text{VT}(D)_i := 0$ )。實際上, 很明顯:

$$m_{i,s} \in D \Leftrightarrow s \leq \text{VT}(D)_i \quad (2)$$

We say that the vector  $\text{VT}(D) = (\text{VT}(D)_i)_{i \in I} \in \mathbb{N}_0^I$  with non-negative components  $\text{VT}(D)_i$  is the *vector time* or *vector timestamp* corresponding to cone  $D$  (cf. [1] or [7] for a more detailed discussion of vector time).

**2.4.6. Partial order on vector timestamps.** 我們引入一個偏序  $\leq$  在所有可能的向量時間  $\mathbb{N}_0^I$  上，這是  $\mathbb{N}_0$  上的傳統順序的乘積：

$$\mathbf{x} = (x_i)_{i \in I} \leq \mathbf{y} = (y_i)_{i \in I} \quad \text{iff} \quad x_i \leq y_i \quad \text{for all } i \in I \quad (3)$$

很明顯，如果且僅如果  $\text{VT}(D) \leq \text{VT}(D')$ ，則  $D \subset D'$ ；因此， $\text{VT}$  是包含在所有消息中的所有錐的集合到  $\mathbb{N}_0^I$  的嚴格保序嵌入。

**2.4.7. Vector timestamp  $\text{Vt}(m)$  of a message  $m$ .** 對於任何消息  $m$ ，我們定義其向量時間戳  $\text{VT}(m)$  為  $\text{VT}(D_m)$ 。換句話說，只有在傳送者  $j$  生成的前  $\text{VT}(m)_j$  個消息被傳遞後，才能傳遞消息  $m$ ，對於所有  $j \in I$  都是如此。

如果  $i$  是消息  $m$  的發送者， $s$  是消息  $m$  的高度，使得  $m = m_{i,s}$ ，那麼  $\text{VT}(m)_i = s - 1$ 。我們可以通過設置  $\text{VT}^+(m)_j = \text{VT}(m)_j$  ( $j \neq i$ )， $\text{VT}^+(m)_i = \text{VT}(m)_i + 1 = s$  定義消息  $m$  的調整向量時間戳  $\text{VT}^+(m)$ 。或者， $\text{VT}^+(m) = \text{VT}(D_m^+)$ ，其中  $D_m^+ := D_m \cup m$  是消息  $m$  的擴展依賴錐（參見 2.4.3）。

注意，對於  $m' \preceq m$ ，當且僅當  $D_{m'}^+ \subset D_m^+$ ，當且僅當  $\text{VT}^+(m') \leq \text{VT}^+(m)$  在  $\mathbb{N}_0^I$  中成立，其中  $m' \preceq m$  的意思是“ $m' \prec m$  或  $m' = m$ ”。同樣地， $m' \prec m$  當且僅當  $D_{m'}^+ \subset D_m$ ，當且僅當  $\text{VT}^+(m') \leq \text{VT}(m)$ 。換句話說，消息（某些或全部）之間的依賴關係  $\prec$  完全由這些消息的調整向量時間戳確定。

**2.4.8. Using vector timestamps to correctly deliver broadcast messages.** 在非拜占庭設置中，向量時間戳可以用於正確傳遞進程組中廣播的消息。<sup>5</sup>具體來說，假設每個廣播消息  $m = m_{i,s}$  包含其發送者  $i$  的索引和此消息的向量時間戳  $\text{VT}(m)$ 。然後，每個接收者  $j$  都知道消息是否可以被傳遞。為此， $j$  保持了到目前為止傳遞的所有消息的錐  $C_j$ ，例如通過維護與  $\text{VT}(C_j)$  相等的當前時間戳  $\text{VT}(j)$ 。換句話說， $\text{VT}(j)_k$  是到目前為止  $j$  處理的發送者  $k$  的消息計數。如果  $\text{VT}(m) \leq \text{VT}(j)$ ，則消息  $m$  立即傳遞，然後  $\text{VT}(j)$  更新為  $\sup(\text{VT}(j), \text{VT}^+(m))$ ；這相當於將消息  $m$  的原始發送者  $i$  的  $\text{VT}(j)_i$  增加一。如果不滿足此條件，則  $m$  可能會被放入等待隊列中，直到  $\text{VT}(j)$  變得足夠大。除了被動地等待所需的廣播消息外， $j$  還可以構建消息索引列表  $(i', s')$ ，這些消息在某個接收但未傳遞的消息  $m$  的  $\text{VT}(m)$  中被隱含提到，並向從中  $j$  得知  $m$  和  $\text{VT}(m)$  的鄰居請求這些消息；另一種策略（實際上是當前 Catchain 協議實現所使用的策略）是定期從隨機選

<sup>5</sup>我們假設進程組中的所有廣播消息都是“因果廣播”或“cbcast”（如 [1] 中的術語），因為我們只需要 cbcast 來實現 Catchain 協議和 Catchain 共識。



擇的鄰居那裡請求這些消息。後一種策略更簡單，因為它不需要記住所有接收到的消息的直接來源（這些來源可能無論如何都不可用）。

### 2.5. Message structure in a catchain. Catchain as a multi-blockchain.

在 Catchain 中，由於需要支持 BFT 協議，消息結構比上述描述複雜一些。特別是，在拜占庭設置中，向量時間戳是不夠的。它們必須通過依賴錐的極大元素的描述來補充（這些描述通常僅在過程組非常大時使用，以至於向量時間戳的大小變得不可行）。

**2.5.1. Describing cones by means of their maximal elements.** 描述消息錐  $D$  的另一種方法（與使用向量時間戳不同）是列出其所有的極大元素  $\text{Max}(D)$ ，即  $m \in D$  且對於任何  $m' \in D$ ，均不滿足  $m \prec m'$ 。當然，為了使這種表示法實用，需要一種適合的方式來引用消息而不完全包含它們。

**2.5.2. Message identifiers inside a catchain.** Catchain 協議使用消息的（適當序列化的）SHA256 哈希作為它們的唯一標識符。如果我們假設 SHA256 沒有碰撞（可在合理的時間內計算，例如多項式時間），那麼消息  $m$  就可以通過其哈希  $\text{SHA256}(m)$  在進程組內被完全識別。

**2.5.3. Message headers.** 在 Catchain 中，一條消息  $m = m_{i,s}$  的標頭（即 Catchain 協議的一個實例）始終包含其發送者的索引  $i$ 、高度  $s$ 、Catchain 識別符（即起源消息的哈希，參見 2.3）和依賴錐的極大元素的哈希集合，即集合  $\text{SHA256}(m'), \text{, } m' \in \text{Max}(D_m)$ 。特別地，由於  $m_{i,s-1} \in \text{Max}(D_m)$ （如果  $s > 1$ ），因此前一條相同發送者的消息  $m_{i,s-1}$  的哈希  $\text{SHA256}(m_{i,s-1})$  總是包含在內；出於性能原因，在消息標頭中還有一個單獨的字段包含  $\text{SHA256}(m_{i,s-1})$ 。如果  $s = 1$ ，那麼就沒有前一條消息，因此使用起源消息的哈希（即 Catchain 識別符，參見 2.3）代替。

消息標頭並不包含向量時間戳  $\text{VT}(m)$ ；然而，標頭隱含地決定了  $\text{VT}(m)$ ，因為：

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

請注意，消息標頭是消息的一部分，特別地，消息的哈希（即消息識別符）取決於標頭中列出的所有數據。因此，我們假設如果 SHA256 沒有已知的碰撞，則消息識別符隱含地確定了相應消息的所有依賴關係。

**2.5.4. Message signatures.** 除此之外，在 Catchain 中的每個消息都由其創建者簽名。由於在 Catchain 中參與的節點（進程）列表事先已知，且該列表包含所有進程的公鑰，因此接收進程可以在接收到消息後立即檢查這些消息的簽名。如果簽名無效，則消息將被丟棄而不進行任何進一步的處理。

**2.5.5. Message encryption.** 在從私有覆蓋網絡中的一個節點傳輸到其鄰居之前，Catchain 中的所有消息都會被加密。然而，這種加密是由低層網絡協議（如 ADNL）執行的，與本文討論的內容無關。我們想提到的是，這裡正確的加密之所以可能是因為參與進程的列表不僅包含所有進程的公鑰，還包括它們的 ADNL 地址（對於網絡傳輸，這些地址實際上是公共加密金鑰）。

需要注意的是，即使沒有加密，這也不會違反協議的 BFT 屬性，因為由於簽名，偽造來自其他發送者的消息是不可能的。然而，這可能會導致對外觀察者的信息泄露，這通常是不可取的。

**2.5.6. Alternative perspective: a catchain as a multi-blockchain.** 需要注意的是，在 Catchain 中由同一發送者  $i$  創建的所有消息都具有一個簡單的“區塊鏈結構”，因為  $m_{i,s+1}$  的標頭包含發送者  $i$  的前一條消息  $m_{i,s}$  的哈希  $\text{SHA256}(m_{i,s})$ （以及來自  $\text{Max}(D_{m_{i,s+1}})$  中其他消息的哈希）。這樣，每個進程  $i$  生成了一個由其消息組成的簡單區塊鏈，其中此區塊鏈的每個“區塊”對應一條消息，並通過其哈希參考前一個區塊，有時還在其區塊中提及其他進程的區塊（即消息）的哈希作為參考。每個區塊都由其創建者簽名。所得到的結構非常類似於 [3, 5] 中考慮的“異步支付通道”的結構，但參與者數量為  $N$  而不是 2。

**2.6. Message propagation in a catchain.** 現在我們可以描述 Catchain 中的消息傳播。具體而言：

- 在 Catchain 下面的（低層次）覆蓋網路協議維護了一個鄰居列表，並為每個鄰居提供 ADNL 通道。這個私有覆蓋網路具有與 Catchain 相同的成員（進程、節點）列表，每個節點的鄰居形成了所有參與節點的（定向）子圖。這個（基本上是隨機的）子圖的強聯通概率非常接近於 1。
- 每個進程不時會生成一些新的消息（根據高層協議的需要）。這些消息按照 2.5.3 中所述的方法增強了 Catchain 消息標頭，簽名後通過覆蓋協議建立的 ADNL 通道傳播到所有已知的鄰居。
- 與通常的簡單覆蓋廣播協議不同，從鄰居那裡接收到的消息不會立即轉發給那些尚未知道它們的副本的所有其他鄰居。相反，首先檢查簽名，並且無效的消息被丟棄。然後，如果它所有的相依消息已經被傳遞，消息就被傳送；否則，消息就被放入等待隊列中。在後一種情況下，標頭中提到的所有所需消息（即集合  $\text{Max}(D_m)$ ）都從發送此消息的鄰居拉取（此外，不時從隨機的鄰居嘗試下載這些缺失的消息）。如果必要，這個過程會遞歸地重複進行，直到一些消息可以被傳送。一

旦消息準備好進行本地傳送（即它的所有相依關係都已經存在），它也會被重新發送到覆蓋網路中的所有鄰居。

- 除了上述的遞歸“拉”機制外，還使用了一種更快的基於向量時間戳的機制，以便可以通過消息的發件人和高度（從接收到的消息的向量時間戳中學習）從鄰居查詢消息。即，每個進程不時向一個隨機選擇的鄰居發送一個包含當前向量時間戳的特殊查詢。這個點對點查詢導致其接收者回傳對發送者未知的所有或一些消息（根據它們的向量時間戳判斷）。
- 一旦從某些發件人發出的消息被檢測到出現“分叉”（例如，在快速或慢速的“拉”過程中，從鄰居那裡學習到了具有相同發件人  $i$  和高度  $s$ ，但哈希值不同的第二個消息），更快的基於向量時間戳的機制就會被禁用。一旦檢測到  $i$  創建的分叉，所有後續向量時間戳的相應分量  $V_{T_i}$  就會被設置為一個特殊值  $\infty$ ，以表示不再有意義比較這些分量的值。
- 當一條消息被傳送（到高層協議）時，這條消息被添加到當前進程處理過的消息錐  $C$  中（並相應地更新當前向量時間戳），並且假定當前進程生成的所有後續消息都依賴於到目前為止傳送的所有消息（即使從高層協議的角度來看這並不是邏輯上必要的）。
- 如果處理的消息錐  $\text{Max}(C)$  的最大元素集合變得太大（包含的元素比 Catchain 的创世消息事先固定的某个数量还多），那么 Catchain 协议会要求高层协议生成一条新消息（如果没有可用的有用有效负载，则为空）。在生成此新消息（并立即将其传递给当前进程）后， $C$  被更新， $\text{Max}(C)$  只包含一个元素（即新消息）。通过这种方式， $\text{Max}(C)$  的大小以及消息头的大小始终保持有限。
- 一旦消息  $m$  被傳送並且集合  $C$  被修改以包含此消息，就會設置一個定時器，在一些小延遲後，會要求高層協議創建一條新消息（如果必要，則為空消息），以便這條新消息  $m^*$  會參照新的  $C$ ，類似於上一項中描述的過程。這個新消息  $m^*$  會被推送給所有鄰居；由於它的標頭包含了新  $C$  的  $\text{Max}(C)$  和  $m \in C$ ，因此鄰居不僅會了解到新生成的消息  $m^*$ ，還會了解到原始接收的消息  $m$ 。如果某些鄰居還沒有  $m$  的副本，它們將需要一個（從當前進程或其他進程）。
- 在 Catchain 中接收和創建的所有（廣播）消息都被存儲在一個特殊的本地數據庫中。這對於新創建的消息尤其重要（參見 3.3.2）：如果一個消息被創建並發送給鄰居，但在創建過程崩潰並重新啟動之前沒

有保存到數據庫（並刷新到磁盤），那麼在重新啟動後可以創建具有相同發件人和高度的另一個消息，進而導致非自願的“分叉”。

**2.7. Forks and their prevention.** 可以看出，上面概述的 Catchain 的多區塊鏈結構（帶有對其他區塊的哈希引用和簽名）對於在建立在 Catchain 上的共識協議（即使用 Catchain 作為在進程組內廣播消息的手段）中的“作弊”幾乎沒有可能性。唯一無法立即檢測到的可能性是創建兩個（或更多）不同版本的同一消息  $m_{i,s}$ （比如， $m'_{i,s}$  和  $m''_{i,s}$ ），並將這個消息的一個版本  $m'_{i,s}$  發送給某些對等節點，另一個版本  $m''_{i,s}$  發送給其他對等節點。如果  $s$  是最小的（對於固定的  $i$ ），那麼這對應於區塊鏈術語中的一個分叉：同一個前一個區塊  $m_{i,s-1}$  的兩個不同的下一個區塊  $m'_{i,s}$  和  $m''_{i,s}$ 。Therefore, the Catchain protocol takes care to detect forks as soon as possible and prevent their propagation.

**2.7.1. Detection of forks.** 檢測分叉很簡單：如果存在兩個不同的區塊  $m'_{i,s}$  和  $m''_{i,s}$ ，它們的創建者  $i \in I$  和高度  $s \geq 1$  相同，並且具有有效的  $i$  的簽名，那麼這就是一個分叉。

**2.7.2. Fork proofs.** Catchain 協議中的區塊簽名是以這樣的方式創建的，即創建分叉證明（即證明進程  $i$  有意創建了一個分叉）特別簡單，因為它是一個非常小的結構（包含一個魔數、 $i$  和  $s$  的值，以及其餘消息的哈希值）的哈希值實際上被簽名。因此，分叉證明只需要兩個這樣的小結構和兩個簽名即可。

**2.7.3. External punishment for creating forks.** 需要注意的是，在 PoS 區塊鏈生成的背景下，可以使用對創建 Catchain 分叉的外部懲罰。即，分叉證明可以提交給一個特殊的智能合約（例如 TON 區塊鏈的選舉人智能合約），自動檢查，並且可能會沒收冒犯方的一部分或全部權益。

**2.7.4. Internal processing of forks.** 一旦檢測到一個分叉（由  $i$  創建），即  $j$  學到了  $i$  創建的兩個不同消息  $m_{i,s}$  和  $m'_{i,s}$ ，並且它們具有相同的高度  $s$ （通常在遞歸下載某些其他消息的相依關係時發生）， $j$  就開始忽略  $i$  和它的所有後續消息。它們不會被接受，也不會進一步廣播。然而，在分叉檢測之前由  $i$  創建的消息，如果在由沒有看到這個分叉的進程創建的消息（區塊）中引用，仍然可以被下載。

**2.7.5. Accepting messages from a “bad” process is bad.** 此外，如果進程  $i$  得知進程  $j$  創建了一個分叉，則  $i$  會通過創建一個包含相應分叉證明的新服務廣播消息（參見 2.7.2）向其鄰居顯示這一信息。此後， $j$  的所有後續消息都不能直接依賴於已知的“壞”的生產者  $i$  的任何消息（但如

果  $k$  在創建引用消息時不知道  $i$  的分叉，則它們仍然可以參考來自另一方  $k$  的消息，該消息直接或間接地引用了  $i$  的消息)。如果  $j$  違反了此限制並創建了具有此類無效引用的消息，那麼這些消息將被該組中的所有誠實進程丟棄。

**2.7.6. The set of “bad” group members is a part of the intrinsic state.** 每個進程  $i$  都保留了其在組中所知的“壞”進程集合的副本，即那些創建了至少一個分叉或違反了 2.7.5 的進程。當  $i$  得知由  $j$  創建的分叉（或  $j$  違反了 2.7.5）時，就會將  $j$  加入到這個集合中；之後，會調用高級協議提供的回調函數。當一個新的廣播消息到達時，就會使用這個集合：如果發送者是壞的，則消息會被忽略並丟棄。

### 3 Block Consensus Protocol

本節我們將解釋 TON 區塊共識協議（參見 1）的基本運作方式，該協議基於通用的 Catchain 協議（參見 2）構建，提供了用於生成和驗證 TON 區塊鏈新區塊的 BFT 協議。TON 區塊共識協議的源代碼位於源代碼樹的 `validator-session` 子目錄中。

**3.1. Internal state of the Block Consensus Protocol.** 高級協議的區塊共識協議引入了一個新的概念到 Catchain 中：區塊共識協議（BCP）的內部狀態，有時也稱為“Catchain 的內部狀態”或簡稱為 *Catchain* 狀態。即，對於每個進程  $i \in I$ ，在 Catchain 協議將一個消息子集（實際上總是一個相依錐） $C_i$  傳遞到高級協議（即在本例中傳遞到區塊共識協議）之後，進程  $i \in I$  會有一個確定的內部狀態  $\sigma_{C_i}$ 。此外，這個狀態  $\sigma_{C_i} = \sigma(C_i)$  只依賴於錐  $C_i$ ，而不依賴於進程  $i \in I$  的身份，並且可以對任何相依錐  $S$ （不一定是某個進程  $i$  在某個時間點上交付的一個錐  $C_i$ ）進行定義。

**3.1.1. Abstract structure of the internal state.** 我們從區塊共識協議使用的內部狀態的抽象結構開始，更具體的細節將在後面提供。

**3.1.2. Updating the internal state.** Catchain 協議對內部狀態一無所知；它只是在傳遞一條消息  $m$  時調用高級協議（即 BCP）提供的適當回調函數。由高級協議來計算新的狀態  $\sigma_{S'}$ ，從先前計算的狀態  $\sigma_S$  和消息  $m$  開始計算，其中  $S' = S \cup m$ （並且必然有  $S \supset D_m$ ，否則  $m$  在此時不能被傳遞）。

**3.1.3. Recursive formula for updating the internal state.** 計算所有相依錐  $S$  的  $\sigma_S$  的抽象設置包括三個組件：

- 初始狀態的值  $\sigma_\emptyset$ （實際上，此值取決於 catchain 的創世區塊；我們在這裡忽略了此依賴關係，因為我們此時只考慮一個 catchain）。
- 一個函數  $f$ ，從先前狀態  $\sigma_{D_m}$  和新傳遞的消息  $m$  計算狀態  $\sigma_{D_m^+}$ ：

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

其中  $D_m$  是消息  $m$  的相依錐， $D_m^+ = D_m \cup m$  是擴展相依錐（參見 2.4.3）。在大多數情況下， $f$  實際上滿足更強的條件：

$$\sigma_{S \cup \{m\}} = f(\sigma_S, m) \quad \text{if } S \text{ and } S \cup \{m\} \text{ are cones and } m \notin S \quad (6)$$

但這個更強的條件不是更新算法所必需的。

- “合併函數”  $g$  從  $\sigma_S$  和  $\sigma_T$  計算  $\sigma_{S \cup T}$ :

$$\sigma_{S \cup T} = g(\sigma_S, \sigma_T) \quad \text{對於任何錐 } S \text{ 和 } T \quad (7)$$

(兩個錐的聯集始終是一個錐)。更新算法只在特定情況下 (即  $T = D_m^+$  且  $m \notin S$ ) 應用此函數  $g$ 。

**3.1.4. Commutativity and associativity of  $g$ .** 請注意, 對於任意的錐  $S$  和  $T$ , (7) 意味着  $g$  的結合律和交換律, 至少在  $g$  應用於可能的狀態 (某個錐  $S$  的形式為  $\sigma_S$  的值) 時是成立的。在這方面,  $g$  在集合  $\Sigma = \sigma_S, \therefore S$  是一個錐上定義了可交換的單調結構。通常情況下,  $g$  在一個更大的狀態值集合  $\tilde{\Sigma}$  上被定義或部分定義, 並且在這個更大的集合  $\tilde{\Sigma}$  上可能是可交換的和結合的, 即對於  $x, y, z \in \tilde{\Sigma}$  (只要等式的兩側都被定義了),  $g(x, y) = g(y, x)$  並且  $g(x, g(y, z)) = g(g(x, y), z)$ , 其中  $\sigma_\emptyset$  是一個單位元, 即對於  $x \in \tilde{\Sigma}$  (在相同的條件下),  $g(x, \sigma_\emptyset) = x = g(\sigma_\emptyset, x)$ 。然而, 這種性質對於共識算法的形式分析很有用, 但對於狀態更新算法來說不是嚴格要求的, 因為該算法使用  $g$  以一種確定性的方式計算  $\sigma_S$ 。

**3.1.5. Commutativity of  $f$ .** 需要注意的是, 如果函數  $f$  滿足更強的條件 (6), 則必須具有可交換性質

$$f(f(\sigma_S, m), m') = f(f(\sigma_S, m'), m) \quad (8)$$

當  $S$  是錐體且  $m$  和  $m'$  是兩個消息, 滿足  $D_m \subset S$ 、 $D_{m'} \subset S$ 、 $m \notin S$  且  $m' \notin S$  時。因為在這種情況下,  $S \cup m$ 、 $S \cup m'$  和  $S \cup m, m'$  也是錐體, 且 (6) 意味著 (8) 兩側均等於  $\sigma_{S \cup m, m'}$ 。類似於 **3.1.4**,  $f$  通常在更大的狀態類似值  $\tilde{\Sigma}$  和消息類似值集合的乘積上被定義或部分定義: 在更大的集合上, 它可能展現出“可交換性”特徵 (8), 也可能不展現。如果它有這個特徵, 這可能對於依賴於  $\sigma_S$  的算法的形式分析是有用的, 但這個特性不是嚴格必要的。

**3.1.6. The state update algorithm.** Catchain (實際上是高級 BCP) 使用  $\sigma_\emptyset$ 、 $f$  和  $g$  來執行狀態更新算法 (由每個進程  $i$  獨立執行), 其運作方式如下:

- 算法跟踪到目前為止已經傳遞的所有消息  $m$  的  $\sigma_{D^+m}$ 。
- 算法跟踪到  $\sigma_{C_i}$ , 其中  $C_i$  是當前依賴錐體, 即所有傳遞給當前進程  $i$  的消息  $m$  的集合。 $\sigma_{C_i}$  的初始值為  $\sigma_\emptyset$ 。

- 當一個新的消息  $m$  被傳遞時，由於  $D_m = \bigcup_{m' \in D_m} D^+m' = \bigcup m' \in \text{Max}(D_m) D^+m'$ ，所以可以通過重複應用  $g$  來計算  $\sigma D^+m$ 。因此，如果  $\text{Max}(D_m) = m'1, \dots, m'k$ ，則

$$\sigma D_m = g\left(\dots g(g(\sigma D^+m'1, \sigma D^+m'2), \sigma D^+m'3), \dots \sigma D^+m'_k\right) \quad (9)$$

在某個固定的順序  $m'_1, \dots, m'_k$  中，消息  $m$  的標題明確地列出了集合  $\text{Max}(D_m)$ ，上述公式是針對此順序進行的（因此計算  $D_m$  是確定性的）。此列表中的第一個元素始終是消息發送方的前一個消息，即如果  $m = m_i, s+1$ ，則  $m'1 = m_i, s$ 。

- 然後，通過應用  $f$ ，計算  $\sigma_{D^+m}$  的值： $\sigma D^+m = f(\sigma D_m, m)$ 。此值被記錄下來以供將來使用。
- 最後，當一個新的消息  $m$  被傳遞給當前進程  $i$ ，從而將  $C_i$  更新為  $C'_i := C_i \cup m$  時，算法使用計算出的值  $\sigma D^+m$  來更新當前狀態

$$\sigma C'_i = g(\sigma C_i, \sigma_{D^+m}) \quad (10)$$

然而，這個狀態是“虛擬的”，因為稍後可能會稍微更改它（尤其是如果  $g$  不是可交換的）。儘管如此，高級算法（BCP）仍然使用它來做出一些重要的決策。

- 當一個新的消息  $m$  被生成並在本地傳遞，從而  $C_i$  變為  $D^+m$ ，則先前計算出的  $\sigma C_i$  的值被丟棄並替換為根據上述通用算法計算出的  $\sigma D^+m$ 。如果  $g$  不是可交換的或不是結合的（例如，可能發生  $g(x, y)$  和  $g(y, x)$  是不同但等效的表示同一個狀態的情況），則這可能會導致稍微更改進程  $i$  的當前“虛擬”狀態。
- 若低層級（catchain）協議向高層級協議報告某個進程  $j \notin i$  是「不良的」（即發現  $j$  已經建立分叉，參見2.7.6，或者  $j$  明知其他進程建立分叉而予以支持，參見2.7.5），那麼當前（虛擬）狀態  $\sigma_{C_i}$  會使用新的集合  $C'_i = \bigcup m \in C_i, m$  由「良好」進程  $k$  建立  $D^+m$  重新計算，並將「合併」函數  $g$  應用於良好進程的最後消息集合（或該集合的極大元素集合）中的  $\sigma D^+_m$  集合。下一個創建的出站消息僅會取決於  $C'_i$  中的消息。

### 3.1.7. Necessity to know the internal state of the other processes.

公式(9)意味著進程  $i$  必須跟踪所有由該進程創建的消息  $m$  的  $\sigma_{D^+m}$ ，以及其他進程創建的消息的  $\sigma D^+m$ 。然而，由於這些內部狀態也是通過更新算法的適當應用來計算的，因此這是可能的。因此，BCP 也會計算和記憶所有的  $\sigma D^+_m$ 。



**3.1.8. Function  $f$  would suffice.** 請注意，更新算法只在  $S$  是包含  $D_m$  但不包含  $m$  的錐時才應用  $g$  來計算  $\sigma_{S \cup D^+ m} = g(\sigma_S, \sigma D^+ m)$ 。因此，每次實際應用  $g$  都可以被滿足擴展性質(6)的  $f$  所替代：

$$\sigma S \cup D^+ m = g(\sigma_S, \sigma D_m^+) = f(\sigma_S, m) \quad (11)$$

但是，更新算法不使用這種「優化」，因為它會禁用下面在3.2.4和3.2.5中描述的更重要的優化。

**3.2. The structure of the internal state.** 內部狀態的結構被優化，以使得(5)中的過渡函數  $f$  和(7)中的合併函數  $g$  能夠盡可能高效地計算，最好不需要可能無限遞迴的過程（僅需要一些迴圈）。這促使將額外的組件納入內部狀態中（即使這些組件可以從內部狀態的其餘部分計算出來），這些組件也必須被存儲和更新。這個包含額外組件的過程類似於使用動態規劃求解問題時所使用的過程，或者類似於通過數學（或結構）歸納來證明陳述的過程。

**3.2.1. The internal state is a representation of a value of an abstract algebraic data type.** 內部狀態的內部表示本質上是一棵（有向）樹（或者說是一個有向無環圖）或者一個節點集合；每個節點包含一些立即的（通常是整數）值和指向其他（之前建立的）節點的幾個指針。如果需要，可以在節點開頭添加一個額外的構造標籤/（一個小整數）來區分幾種可能性。這個結構非常類似於用於表示抽象代數數據類型值的結構化編程語言（如 Haskell）中的結構。

**3.2.2. The internal state is persistent.** 內部狀態是持久的，意味著在 catchain 活動期間，用於分配內部狀態中的節點的內存永遠不會被釋放。此外，catchain 的內部狀態實際上是分配在一個巨大的連續內存緩衝區內，新節點總是在該緩衝區已使用部分的末尾進行分配，並通過移動指針進行。這樣，該緩衝區內一個節點對其他節點的引用可以由該節點與緩衝區開始處的整數偏移量來表示。每個內部狀態都是由一個指向其在此緩衝區內的根節點的指針來表示的；這個指針也可以用從緩衝區開始處的整數偏移量來表示。

**3.2.3. The internal state of a catchain is flushed to an append-only file.** 根據上述用於存儲 catchain 內部狀態的緩衝區結構，其結果是只有在其末尾添加一些新數據時才會更新。這意味著 catchain 的內部狀態（或者說包含所有所需內部狀態的緩衝區）可以刷新到僅追加的文件中，在重新啟動後很容易恢復。在重新啟動之前唯一需要存儲的其他數據是 catchain 當前狀態的偏移量（從緩衝區開始處或該文件的開始處測量）。可以使用簡單的鍵值數據庫來實現這個目的。

**3.2.4. Sharing data between different states.** 結果發現，表示新狀態  $\sigma_{S \cup m} = f(\sigma_S, m)$  的樹（或者說是 dag）與先前狀態  $\sigma_S$  共享大的子樹，同樣地， $\sigma_{S \cup T} = g(\sigma_S, \sigma_T)$  與  $\sigma_S$  和  $\sigma_T$  共享大的子樹。在 BCP 中用於表示狀態的持久結構使得可以重複使用相同的指針來表示這些共享的數據結構，而不是將它們複製一份。

**3.2.5. Memoizing nodes.** 在計算新狀態（即函數  $f$  的值）時，使用的另一種技術是記憶化新節點，這也是從功能編程語言中借鑒過來的。具體而言，每當構造一個新節點（在包含特定 catchain 所有狀態的巨大緩衝區內部），都會計算它的哈希值，並使用簡單的哈希表來查找具有相同哈希值的最新節點。如果找到具有相同內容的該哈希值的節點，則新構造的節點將被丟棄，而返回對具有相同內容的舊節點的引用。另一方面，如果沒有找到新節點的副本，那麼哈希表將被更新，緩衝區（分配）指針將被推進，並將新節點的指針返回給調用者。

這樣，即使不直接通過應用函數  $f$  如 3.2.4 中所述相關，如果不同的進程最終進行相似的計算並具有相似的狀態，那麼這些狀態的大部分將被共享。

**3.2.6. Importance of optimization techniques.** 用於在同一個 catchain 內部共享不同內部狀態的部分的優化技術 3.2.4 和 3.2.5 對於改善 BCM 在大型進程組中的內存占用和性能具有極其重要的作用。在  $N \approx 100$  的進程組中，改進的幅度可以達到數個數量級。如果沒有這些優化，BCM 將無法適用於其預定的目的（在 TON 區塊鏈中由驗證器生成新塊的 BFT 共識）。

**3.2.7. Message  $m$  contains a hash of state  $\sigma_{D_m^+}$ .** 每個消息  $m$  都包含相應狀態  $\sigma_{D_m^+}$  的（Merkle）哈希值。粗略地說，使用 3.2.1 中的節點樹表示遞歸地計算此哈希值：節點內的所有節點引用都被替換為所引用節點的（遞歸計算的）哈希值，然後計算結果字節序列的簡單 64 位哈希值。這個哈希值也用於如 3.2.5 所述的記憶化。

消息中此字段的目的是對不同進程（以及可能是狀態更新算法的不同實現）對  $\sigma_{D^+m}$  的計算進行健全性檢查：一旦為新交付的消息  $m$  計算了  $\sigma_{D^+m}$ ，則比較計算出的  $\sigma_{D_m^+}$  的哈希值與  $m$  的標頭中存儲的值是否相等。如果這些值不相等，則在錯誤日誌中輸出錯誤消息（軟件不再執行其他操作）。可以檢查這些錯誤日誌以檢測 BCP 不同版本之間的錯誤或不兼容性。

**3.3. State recovery after restart or crashes.** Catchain 通常由 BCP 使用幾分鐘；在此期間，運行 Catchain 協議的程序（驗證器軟件）可能會被終止並重新啟動，無論是故意（例如，因為計劃中的軟件更新）還是意外（該程序可能會因此或某些其他子系統中的錯誤而崩潰，然後重新啟

動)。處理這種情況的一種方法是忽略所有在上次重新啟動之後未創建的 catchain。但是，這將導致一些驗證器在幾分鐘內不參與任何塊的創建（直到創建下一個 catchain 實例），這是不希望的。因此，在每次重新啟動後，運行 catchain 狀態恢復協議，以便驗證器可以繼續參與同一個 catchain。

**3.3.1. Database of all delivered messages.** 為此，為每個活動的 catchain 創建一個特殊的數據庫。此數據庫包含所有已知和傳遞的消息，按其標符（哈希值）進行索引。為此，一個簡單的鍵值數據庫就足夠了。還將存儲在此數據庫中的當前進程  $i$  生成的最新出站消息  $m = m_{i,s}$  的哈希值。重新啟動後，所有直到  $m$  的消息都按正確的順序遞歸性地傳遞（就像如果這些消息剛從網絡以任意順序接收到一樣），並由更高級別的協議進行處理，直到  $m$  最終被傳遞，從而恢復當前狀態。

**3.3.2. Flushing new messages to disk.** 如2.6所述，新創建的消息存儲在所有已傳遞消息的數據庫中（參見3.3.1），並在將新消息發送到所有網絡鄰居之前將數據庫刷新到磁盤上。通過這種方式，我們可以確保如果系統崩潰並重新啟動，消息不會丟失，從而避免意外創建分支。

**3.3.3. Avoiding the recomputation of states  $\sigma_{D_m^+}$ .** 實現可以使用包含所有先前計算狀態的追加文件，如3.2.3所述，以避免在重新啟動後重新計算所有狀態，以犧牲磁盤空間換取計算能力。然而，當前的實現沒有使用這種優化。

**3.4. High-level description of Block Consensus Protocol.** 現在我們準備介紹 TON 區塊鏈驗證器使用的區塊鏈共識協議的高級描述，以生成並在新區塊鏈塊上實現共識。基本上，這是一個三階段提交協議，運行在一個 Catchain 上（Catchain 協議的一個實例），它被用作一個在一個進程組中“硬化”的消息廣播系統。

**3.4.1. Creation of new catchain messages.** 回想一下，低級 Catchain 協議不會自己創建廣播消息（唯一的例外是具有分叉證明的服務廣播，參見2.7.5）。相反，當需要創建新消息時，通過調用回調來請求高級協議（BCP）進行此操作。此外，新消息的創建可能會受到當前虛擬狀態的變化和定時器警報的觸發。

**3.4.2. Payload of catchain messages.** 這樣，catchain 消息的有效負載始終由更高層級的協議（例如 BCP）確定。對於 BCP，該有效負載包括以下內容：

- 當前的 Unix 時間。在同一進程的後續消息中，它必須不斷增加。（如果違反此限制，處理此消息的所有進程將默認地將此 Unix 時間替換為相同發送者的先前消息中看到的最大 Unix 時間。）

- 一個或多個 *BCP* 事件，可以是以下任一類型。

**3.4.3. BCP events.** 我們剛剛解釋了 *catchain* 消息的有效負載包含幾個（可能為零）*BCP* 事件。現在我們列出所有允許的 *BCP* 事件類型。

- $\text{SUBMIT}(\text{round}, \text{candidate})$  — suggest a new block candidate
- $\text{APPROVE}(\text{round}, \text{candidate}, \text{signature})$  — a block candidate has passed local validation
- $\text{REJECT}(\text{round}, \text{candidate})$  — a block candidate has failed local validation
- $\text{COMMITSIGN}(\text{round}, \text{candidate}, \text{signature})$  — a block candidate has been accepted and signed
- $\text{VOTE}(\text{round}, \text{candidate})$  — a vote for a block candidate
- $\text{VOTEFOR}(\text{round}, \text{candidate})$  — this block candidate must be voted for in this round (even if the current process has another opinion)
- $\text{PRECOMMIT}(\text{round}, \text{candidate})$  — a preliminary commitment to a block candidate (used in three-phase commit scheme)

**3.4.4. Protocol parameters.** Several parameters of *BCP* must be fixed in advance (in the genesis message of the *catchain*, where they are initialized from the values of the configuration parameters extracted from the current masterchain state):

- $K$  — duration of one attempt (in seconds). It is an integer amount of seconds in the current implementation; however, this is an implementation detail, not a restriction of the protocol
- $Y$  — number of *fast* attempts to accept a candidate
- $C$  — block candidates suggested during one round
- $\Delta_i$  for  $1 \leq i \leq C$  — delay before suggesting the block candidate with priority  $i$
- $\Delta_\infty$  — delay before approving the null candidate

Possible values for these parameters are  $K = 8$ ,  $Y = 3$ ,  $C = 2$ ,  $\Delta_i = 2(i - 1)$ ,  $\Delta_\infty = 2C$ .

**3.4.5. Protocol overview.** The BCP consists of several *rounds* that are executed inside the same catchain. More than one round may be active at one point of time, because some phases of a round may overlap with other phases of other rounds. Therefore, all BCP events contain an explicit round identifier *round* (a small integer starting from zero). Every round is terminated either by (collectively) accepting a *block candidate* suggested by one of the participating processes, or by accepting a special *null candidate*—a dummy value indicating that no real block candidate was accepted, for example because no block candidates were suggested at all. After a round is terminated (from the perspective of a participating process), i.e., once a block candidate collects COMMITSIGN signatures of more than  $2/3$  of all validators, only COMMITSIGN events may be added to that round; the process automatically starts participating in the next round (with the next identifier) and ignores all BCP events with different values of *round*.<sup>6</sup>

Each round is subdivided into several *attempts*. Each attempt lasts a predetermined time period of  $K$  seconds (BCP uses clocks to measure time and time intervals and assumes that clocks of “good” processes are more or less in agreement with each other; therefore, BCP is not an asynchronous BFT protocol). Each attempt starts at Unixtime exactly divisible by  $K$  and lasts for  $K$  seconds. The attempt identifier *attempt* is the Unixtime of its start divided by  $K$ . Therefore, the attempts are numbered more or less consecutively by 32-bit integers, but not starting from zero. The first  $Y$  attempts of a round are *fast*; the remaining attempts are *slow*.

**3.4.6. Attempt identification. Fast and slow attempts.** In contrast with rounds, BCP events do not have a parameter to indicate the attempt they belong to. Instead, this attempt is implicitly determined by the Unix time indicated in the payload of the catchain message containing the BCP event (cf. 3.4.2). Furthermore, the attempts are subdivided into *fast* (the first  $Y$  attempts of a round in which a process takes part) and *slow* (the subsequent attempts of the same round). This subdivision is also implicit: the first BCP event sent by a process in a round belongs to a certain attempt, and  $Y$  attempts starting from this one are considered fast by this process.

**3.4.7. Block producers and block candidates.** There are  $C$  designated block producers (member processes) in each round. The (ordered) list of these

---

<sup>6</sup>This also means that each process implicitly determines the Unixtime of the start of the next round, and computes all delays, e.g., the block candidate submission delays, starting from this time.

block producers is computed by a deterministic algorithm (in the simplest case, processes  $i, i+1, \dots, i+C-1$  are used in the  $i$ -th round, with the indices taken modulo  $N$ , the total number of processes in the catchain) and is known to all participants without any extra communication or negotiation. The processes are ordered in this list by decreasing priority, so the first member of the list has the highest priority (i.e., if it suggests a block candidate in time, this block candidate has a very high chance to be accepted by the protocol).

The first block producer may suggest a block candidate immediately after the round starts. Other block producers can suggest block candidates only after some delay  $\Delta_i$ , where  $i$  is the index of the producer in the list of designated block producers, with  $0 = \Delta_1 \leq \Delta_2 \leq \dots$ . After some predetermined period of time  $\Delta_\infty$  elapses from the round start, a special *null candidate* is assumed automatically suggested (even if there are no explicit BCP events to indicate this). Therefore, at most  $C + 1$  block candidates (including the null candidate) are suggested in a round.

**3.4.8. Suggesting a block candidate.** A block candidate for the TON Blockchain consists of two large “files” — the block and the collated data, along with a small header containing the description of the block being generated (most importantly, the complete *block identifier* for the block candidate, containing the workchain and the shard identifier, the block sequence number, its file hash and its root hash) and the SHA256 hashes of the two large files. Only a part of this small header (including the hashes of the two files and other important data) is used as *candidate* in BCP events such as SUBMIT or COMMITSIGN to refer to a specific block candidate. The bulk of the data (most importantly, the two large files) is propagated in the overlay network associated with the catchain by the streaming broadcast protocol implemented over ADNL for this purpose (cf. [3, 5]). This bulk data propagation mechanism is unimportant for the validity of the consensus protocol (the only important point is that the hashes of the large files are part of BCP events and hence of the catchain messages, where they are signed by the sender, and these hashes are checked after the large files are received by any participating nodes; therefore, nobody can replace or corrupt these files). A SUBMIT(*round, candidate*) BCP event is created in the catchain by the block producer in parallel with the propagation of the block candidate, indicating the submission of this specific block candidate by this block producer.

**3.4.9. Processing block candidates.** Once a process observes a SUBMIT BCP event in a delivered catchain message, it checks the validity of this event (for instance, its originating process must be in the list of designated producers, and current Unixtime must be at least the start of the round plus the minimum delay  $\Delta_i$ , where  $i$  is the index of this producer in the list of designated producers), and if it is valid, remembers it in the current catchain state (cf. 3.1). After that, when a streaming broadcast containing the files associated with this block candidates (with correct hash values) is received (or immediately, if these files are already present), the process invokes a validator instance to validate the new block candidate (even if this block candidate was suggested by this process itself!). Depending on the result of this validation, either an APPROVE(*round, candidate, signature*) or a REJECT(*round, candidate*) BCP event is created (and embedded into a new catchain message). Note that the *signature* used in APPROVE events uses the same private key that will ultimately be used to sign the accepted block, but the signature itself is different from that used in COMMITSIGN (the hash of a structure with different magic number is actually signed). Therefore, this interim signature cannot be used to fake the acceptance of this block by this particular validator process to an outside observer.

**3.4.10. Overview of one round.** Each round of BCP proceeds as follows:

- At the beginning of a round, several processes (from the predetermined list of designated producers) submit their block candidates (with certain delays depending on their producer priority) and reflect this fact by means of SUBMIT events (incorporated into catchain messages).
- Once a process receives a submitted block candidate (i.e., observes a SUBMIT event and receives all necessary files by means external to the consensus protocol), it starts the validation of this candidate and eventually creates either an APPROVE or a REJECT event for this block candidate.
- During each *fast attempt* (i.e., one of the first  $Y$  attempts) every process votes either for a block candidate that has collected the votes of more than  $2/3$  of all processes, or, if there are no such candidates yet, for the valid (i.e., APPROVED by more than  $2/3$  of all processes) block candidate with the highest priority. The voting is performed by means of creating VOTE events (embedded into new catchain messages).

- During each *slow attempt* (i.e., any attempt except the first  $Y$ ) every process votes either for a candidate that was PRECOMMITTED before (by the same process), or for a candidate that was suggested by VOTEFOR.
- If a block candidate has received votes from more than  $2/3$  of all processes during the current attempt, and the current process observes these votes (which are collected in the catchain state), a PRECOMMIT event is created, indicating that the process will vote only for this candidate in future.
- If a block candidate collects PRECOMMITs from more than  $2/3$  of all processes inside an attempt, then it is assumed to be accepted (by the group), and each process that observes these PRECOMMITs creates a COMMITSIGN event with a valid block signature. These block signatures are registered in the catchain, and are ultimately collected to create a “block proof” (containing signatures of more than  $2/3$  of the validators for this block). This block proof is the external output of the consensus protocol (along with the block itself, but without its collated data); it is ultimately propagated in the overlay network of all full nodes that have subscribed to new blocks of this shard (or of the masterchain).
- Once a block candidate collects COMMITSIGN signatures from more than  $2/3$  of all validators, the round is considered finished (at least from the perspective of a process that observes all these signatures). After that, only a COMMITSIGN can be added to that round by this process, and the process automatically starts participating in the next round (and ignores all events related to other rounds).

Note that the above protocol may lead to a validator signing (in a COMMITSIGN event) a block candidate that was REJECTED by the same validator before (this is a kind of “submitting to the will of majority”).

**3.4.11. Vote and PreCommit messages are created deterministically.** Note that each process can create at most one VOTE and at most one PRECOMMIT event in each attempt. Furthermore, these events are completely determined by the state  $\sigma_{D_m}$  of the sender of catchain message  $m$  containing such an event. Therefore, the receiver can detect invalid VOTE or PRECOMMIT events and ignore them (thus mitigating byzantine behavior of



other participants). On the other hand, a message  $m$  that should contain a VOTE or a PRECOMMIT event according to the corresponding state  $\sigma_{D_m}$  but does not contain one can be received. In this case, the current implementation automatically creates missing events and proceeds as if  $m$  had contained them from the very beginning. However, such instances of byzantine behavior are either corrected or ignored (and a message is output into the error log), but the offending processes are not otherwise punished (because this would require very large misbehavior proofs for outside observers that do not have access to the internal state of the catchain).

**3.4.12. Multiple Votes and PreCommits of the same process.** Note that a process usually ignores subsequent VOTES and PRECOMMITs generated by the same originating process inside the same attempt, so normally a process can vote for at most one block candidate. However, it may happen that a “good” process indirectly observes a fork created by a byzantine process, with VOTES for different block candidates in different branches of this fork (this can happen if the “good” process learns about these two branches from two other “good” processes that did not see this fork before). In this case, both VOTES (for different candidates) are taken into account (added into the merged state of the current process). A similar logic applies to PRECOMMITs.

**3.4.13. Approving or rejecting block candidates.** Notice that a block candidate cannot be APPROVED or REJECTED before it has been SUBMITTED (i.e., an APPROVE event that was not preceded by a corresponding SUBMIT event will be ignored), and that a candidate cannot be approved before the minimum time of its submission (the round start time plus the priority-dependent delay  $\Delta_i$ ) is reached, i.e., any “good” process will postpone the creation of its APPROVE until this time. Furthermore, one cannot APPROVE more than one candidate of the same producer in the same round (i.e., even if a process SUBMITS several candidates, only one of them—presumably the first one—will be APPROVED by other “good” processes; as usual, this means that subsequent APPROVE events will be ignored by “good” processes on receipt).

**3.4.14. Approving the null block candidate.** The implicit null block candidate is also explicitly approved (by creating an APPROVE event) by all (good) processes, once the delay  $\Delta_\infty$  from the start of the round expires.

**3.4.15. Choosing a block candidate for voting.** Each process chooses one of the available block candidates (including the implicit null candidate) and votes for this candidate (by creating a VOTE event) by applying the following rules (in the order they are presented):

- If the current process created a PRECOMMIT event for a candidate during one of the previous attempts, and no other candidate has collected votes from more than  $2/3$  of all processes since (i.e., inside one of the subsequent attempts, including the current one so far; we say that the PRECOMMIT event is still *active* in this case), then the current process votes for this candidate again.
- If the current attempt is fast (i.e., one of the first  $Y$  attempts of a round from the perspective of the current process), and a candidate has collected votes from more than  $2/3$  of all processes during the current or one of the previous attempts, the current process votes for this candidate. In the case of a tie, the candidate from the latest of all such attempts is chosen.
- If the current attempt is fast, and the previous rules do not apply, then the process votes for the candidate with the highest priority among all *eligible candidates*, i.e., candidates that have collected APPROVES (observable by the current process) from more than  $2/3$  of all processes.
- If the current attempt is slow, then the process votes only after it receives a valid VOTEFOR event in the same attempt. If the first rule is applicable, the process votes according to it (i.e., for the previously PRECOMMITTED candidate). Otherwise it votes for the block candidate that is mentioned in the VOTEFOR event. If there are several such valid events (during the current attempt), the candidate with the smallest hash is selected (this may happen in rare situations related to different VOTEFOR events created in different branches of a fork, cf. 3.4.12).

The “null candidate” is considered to have the least priority. It also requires an explicit APPROVE before being voted for (with the exception of the first two rules).

**3.4.16. Creating VoteFor events during slow attempts.** A VOTEFOR event is created at the beginning of a slow attempt by the *coordinator* — the process with index  $attempt \bmod N$  in the ordered list of all processes

participating in the catchain (as usual, this means that a VOTEFOR created by another process will be ignored by all “good” processes). This VOTEFOR event refers to one of the block candidates (including the null candidate) that have collected APPROVES from more than  $2/3$  of all processes, usually randomly chosen among all such candidates. Essentially, this is a suggestion to vote for this block candidate directed to all other processes that do not have an active PRECOMMIT.

**3.5. Validity of BCP.** Now we present a sketch of the proof of validity of TON Block Consensus Protocol (BCP) described above in 3.4, assuming that less than one third of all processes exhibit byzantine (arbitrarily malicious, possibly protocol-violating) behavior, as it is customary for Byzantine Fault Tolerant protocols. During this subsection, we consider only one round of BCP, subdivided into several attempts.

**3.5.1. Fundamental assumption.** Let us emphasize once again that we assume that *less than one third of all processes are byzantine*. All other processes are assumed to be *good*, i.e., they follow the protocol.

**3.5.2. Weighted BCP.** The reasoning in this subsection is valid for the *weighted variant of BCP* as well. In this variant, each process  $i \in I$  is pre-assigned a positive weight  $w_i > 0$  (fixed in the genesis message of the catchain), and statements about “more than  $2/3$  of all processes” and “less than one third of all processes” are understood as “more than  $2/3$  of all processes by weight”, i.e., “a subset  $J \subset I$  of processes with total weight  $\sum_{j \in J} w_j > \frac{2}{3} \sum_{i \in I} w_i$ ”, and similarly for the second property. In particular, our “fundamental assumption” 3.5.1 is to be understood in the sense that “the total weight of all byzantine processes is less than one third of the total weight of all processes”.

**3.5.3. Useful invariants.** We collect here some useful invariants obeyed by all BCP events during one round of BCP (inside a catchain). These invariants are enforced in two ways. Firstly, any “good” (non-byzantine) process will not create events violating these invariants. Secondly, even if a “bad” process creates an event violating these invariants, all “good” processes will detect this when a catchain message containing this event is delivered to BCP and ignore such events. Some possible issues related to forks (cf. 3.4.12) remain even after these precautions; we indicate how these issues are resolved separately, and ignore them in this list. So:

- There is at most one SUBMIT event by each process (inside one round of BCP).
- There is at most one APPROVE or REJECT event by each process related to one candidate (more precisely, even if there are multiple candidates created by the same designated block producer, only one of them can be APPROVED by another process).<sup>7</sup> This is achieved by requiring all “good” processes to ignore (i.e., not to create APPROVES or REJECTS for) all candidates suggested by the same producer but the very first one they have learned about.
- There is at most one VOTE and at most one PRECOMMIT event by each process during each attempt.
- There is at most one VOTEFOR event during each (slow) attempt.
- There is at most one COMMITSIGN event by each process.
- During a slow attempt, each process votes either for its previously PRECOMMITTED candidate, or for the candidate indicated in the VOTEFOR event of this attempt.

One might somewhat improve the above statements by adding the word “valid” where appropriate (e.g., there is at most one *valid* SUBMIT event. . .).

#### 3.5.4. More invariants.

- There is at most one eligible candidate (i.e., candidate that has received APPROVES from more than  $2/3$  of all processes) from each designated producer, and no eligible candidates from other producers.
- There are at most  $C + 1$  eligible candidates in total (at most  $C$  candidates from  $C$  designated producers, plus the null candidate).

---

<sup>7</sup>In fact, REJECTS appear only in this restriction, and do not affect anything else. Therefore, any process can abstain from sending REJECTS without violating the protocol, and REJECT events could have been removed from the protocol altogether. Instead, the current implementation of the protocol still generates REJECTS, but does not check anything on their receipt and does not remember them in the catchain state. Only a message is output into the error log, and the offending candidate is stored into a special directory for future study, because REJECTS usually indicate either the presence of a byzantine adversary, or a bug in the collator (block generation) or validator (block verification) software either on the node that suggested the block or on the node that created the REJECT event.

- A candidate may be accepted only if it has collected more than  $2/3$  PRECOMMITs during the same attempt (more precisely, a candidate is accepted only if there are PRECOMMIT events created by more than  $2/3$  of all processes for this candidate and belonging to the same attempt).
- A candidate may be VOTED for, PRECOMMITTED, or mentioned in a VOTEFOR only if it is an *eligible candidate*, meaning that it has previously collected APPROVES from more than  $2/3$  of all validators (i.e., a valid VOTE event may be created for a candidate only if APPROVE events for this candidate have been previously created by more than  $2/3$  of all processes and registered in catchain messages observable from the message containing the VOTE event, and similarly for PRECOMMIT and VOTEFOR events).

**3.5.5. At most one block candidate is accepted.** Now we claim that *at most one block candidate can be accepted (in a round of BCP)*. Indeed, a candidate can be accepted only if it collects PRECOMMITs from more than  $2/3$  of all processes inside the same attempt. Therefore, two different candidates cannot achieve this during the same attempt (otherwise more than one third of all validators must have created PRECOMMITs for two different candidates inside an attempt, thus violating the above invariants; but we have assumed that less than one third of all validators exhibit byzantine behavior). Now suppose that two different candidates  $c_1$  and  $c_2$  have collected PRECOMMITs from more than  $2/3$  of all processes in two different attempts  $a_1$  and  $a_2$ . We may assume that  $a_1 < a_2$ . According to the first rule of **3.4.15**, each process that has created a PRECOMMIT for  $c_1$  during attempt  $a_1$  must continue voting for  $c_1$  in all subsequent attempts  $a' > a_1$ , or at least cannot vote for any other candidate, unless another candidate  $c'$  collects VOTES of more than  $2/3$  of all processes during a subsequent attempt (and this invariant is enforced even if some processes attempt not to create these new VOTE events for  $c_1$ , cf. **3.4.11**). Therefore, if  $c_2 \neq c_1$  has collected the necessary amount of PRECOMMITs during attempt  $a_2 > a_1$ , there is at least one attempt  $a'$ ,  $a_1 < a' \leq a_2$ , such that some  $c' \neq c_1$  (not necessarily equal to  $c_2$ ) has collected VOTES of more than  $2/3$  of all processes during attempt  $a'$ . Let us fix the smallest such  $a'$ , and the corresponding  $c' \neq c_1$  that has collected many votes during attempt  $a'$ . More than  $2/3$  of all validators have voted for  $c'$  during attempt  $a'$ , and more than  $2/3$  of all validators have PRECOMMITTED for  $c_1$  during attempt  $a_1$ , and by the minimality of  $a'$  there

was no attempt  $a''$  with  $a_1 < a'' < a'$ , such that a candidate distinct from  $c_1$  collected more than  $2/3$  of all votes during attempt  $a''$ . Therefore, all validators that PRECOMMITTED for  $c_1$  could vote only for  $c_1$  during attempt  $a'$ , and at the same time we supposed that  $c'$  has collected votes from more than  $2/3$  of all validators during the same attempt  $a'$ . This implies that more than  $1/3$  of all validators have somehow voted both for  $c_1$  and  $c'$  during this attempt (or voted for  $c'$  while they could have voted only for  $c_1$ ), i.e., more than  $1/3$  of all validators have exhibited byzantine behavior. This is impossible by our fundamental assumption **3.5.1**.

**3.5.6. At most one block candidate may be PreCommitted during one attempt.** Note that all valid PRECOMMIT events (if any) created inside the same attempt must refer to the same block candidate, by the same reasoning as in the first part of **3.5.5**: since a valid PRECOMMIT event for a candidate  $c$  may be created only after votes from more than  $2/3$  of all processes are observed for this candidate inside the same attempt (and invalid PRECOMMITs are ignored by all good processes), the existence of valid PRECOMMIT events for different candidates  $c_1$  and  $c_2$  inside the same attempt would imply that more than one third of all processes have voted both for  $c_1$  and  $c_2$  inside this attempt, i.e., they have exhibited byzantine behavior. This is impossible in view of our fundamental assumption **3.5.1**.

**3.5.7. A previous PreCommit is deactivated by the observation of a newer one.** We claim that *whenever a process with an active PRECOMMIT observes a valid PRECOMMIT created by any process in a later attempt for a different candidate, its previously active PRECOMMIT is deactivated*. Recall that we say that a process has an *active PRECOMMIT* if it has created a PRECOMMIT for a certain candidate  $c$  during a certain attempt  $a$ , did not create any PRECOMMIT during any attempts  $a' > a$ , and did not observe votes of more than  $2/3$  of all validators for any candidate  $\neq c$  during any attempts  $a' > a$ . Any process has at most one active PRECOMMIT, and if it has one, it must vote only for the precommitted candidate.

Now we see that if a process with an active PRECOMMIT for a candidate  $c$  since attempt  $a$  observes a valid PRECOMMIT (usually by another process) for a candidate  $c'$  created during some later attempt  $a' > a$ , then the first process must also observe all dependencies of the message that contains the newer PRECOMMIT; these dependencies necessarily include valid VOTES from more than  $2/3$  of all validators for the same candidate  $c' \neq c$  created during the same attempt  $a' > a$  (because otherwise the newer PRECOMMIT

would not be valid, and would be ignored by the other process); by definition, the observation of all these VOTES deactivates the original PRECOMMIT.

### 3.5.8. Assumptions for proving the convergence of the protocol.

Now we are going to prove that the protocol described above *converges* (i.e., terminates after accepting a block candidate) with probability one under some assumptions, which essentially tell us that there are enough “good” processes (i.e., processes that diligently follow the protocol and do not introduce arbitrary delays before sending their new messages), and that these good processes enjoy good network connectivity at least from time to time. More precisely, our assumptions are as follows:

- There is a subset  $I^+ \subset I$  consisting of “good” processes and containing more than  $2/3$  of all processes.
- All processes from  $I^+$  have well-synchronized clocks (differing by at most  $\tau$ , where  $\tau$  is a bound for network latency described below).
- If there are infinitely many attempts, then infinitely many attempts are “good” with respect to network connectivity between processes from  $I^+$ , meaning that all messages created by a process from  $I^+$  during this attempt or earlier are delivered to any other process from  $I^+$  within at most  $\tau > 0$  seconds after being created with probability at least  $q > 0$ , where  $\tau > 0$  and  $0 < q < 1$  are some fixed parameters, such that  $5\tau < K$ , where  $K$  is the duration of one attempt.
- Furthermore, if the protocol runs for infinitely many attempts, then any arithmetic progression of attempts contains infinitely many “good” attempts in the sense described above.
- A process from  $I^+$  creates a VOTEFOR during a slow attempt after some fixed or random delay after the start of the slow attempt, in such a way that this delay belongs to the interval  $(\tau, K - 3\tau)$  with probability at least  $q'$ , where  $q' > 0$  is a fixed parameter.
- A process from  $I^+$ , when it is its turn to be the coordinator of a slow attempt, chooses a candidate for VOTEFOR uniformly at random among all eligible candidates (i.e., those candidates that have collected APPROVES from more than  $2/3$  of all validators).

**3.5.9. The protocol terminates under these assumptions.** Now we claim that *(each round of) the BCP protocol as described above terminates with probability one under the assumptions listed in 3.5.8.* The proof proceeds as follows.

- Let us assume that the protocol does not converge. Then it continues running forever. We are going to ignore the first several attempts, and consider only attempts  $a_0, a_0 + 1, a_0 + 2, \dots$  starting from some  $a_0$ , to be chosen later.
- Since all processes from  $I^+$  continue participating in the protocol, they will create at least one message not much later than the start of the round (which may be perceived slightly differently by each process). For instance, they will create an APPROVE for the null candidate no later than  $\Delta_\infty$  seconds from the start of the round. Therefore, they will consider all attempts slow at most  $KY$  seconds afterwards. By choosing  $a_0$  appropriately, we can assume that all attempts we consider are slow from the perspective of all processes from  $I^+$ .
- After a “good” attempt  $a \geq a_0$  all processes from  $I^+$  will see the APPROVES for the null candidate created by all other processes from  $I^+$ , and will deem the null candidate eligible henceforth. Since there are infinitely many “good” attempts, this will happen sooner or later with probability one. Therefore, we can assume (increasing  $a_0$  if necessary) that there is at least one eligible candidate from the perspective of all processes from  $I^+$ , namely, the null candidate.
- Furthermore, there will be infinitely many attempts  $a \geq a_0$  that are perceived slow by all processes from  $I^+$ , that have a coordinator from  $I^+$ , and that are “good” (with respect to the network connectivity) as defined in 3.5.8. Let us call such attempts “very good”.
- Consider one “very good” slow attempt  $a$ . With probability  $q' > 0$ , its coordinator (which belongs to  $I^+$ ) will wait for  $\tau' \in (\tau, K - 3\tau)$  seconds before creating its VOTEFOR event. Consider the most recent PRECOMMIT event created by any process from  $I^+$ ; let us suppose it was created during attempt  $a' < a$  for some candidate  $c'$ . With probability  $qq' > 0$ , the catchain message carrying this PRECOMMIT will be already delivered to the coordinator at the time of generation of



its VOTEFOR event. In that case, the catchain message carrying this VOTEFOR will depend on this PRECOMMIT( $c'$ ) event, and all “good” processes that observe this VOTEFOR will also observe its dependencies, including this PRECOMMIT( $c'$ ). We see that *with probability at least  $qq'$ , all processes from  $I^+$  that receive the VOTEFOR event during a “very good” slow attempt receive also the most recent PRECOMMIT (if any).*

- Next, consider any process from  $I^+$  that receives this VOTEFOR, for a randomly chosen eligible candidate  $c$ , and suppose that there are already some PRECOMMITs, and that the previous statement holds. Since there are at most  $C + 1$  eligible candidates (cf. 3.5.4), with probability at least  $1/(C + 1) > 0$  we’ll have  $c = c'$ , where  $c'$  is the most recently PRECOMMITTED candidate (there is at most one such candidate by 3.5.6). In this case, all processes from  $I^+$  will vote for  $c = c'$  during this attempt immediately after they receive this VOTEFOR (which will be delivered to any process  $j \in I^+$  less than  $K - 2\tau$  seconds after the beginning of the attempt with probability  $qq'$ ). Indeed, if a process  $j$  from  $I^+$  did not have an active PRECOMMIT, it will vote for the value indicated in VOTEFOR, which is  $c$ . If  $j$  had an active PRECOMMIT, and it is as recent as possible, i.e., also created during attempt  $a'$ , then it must have been a PRECOMMIT for the same value  $c' = c$  (because we know about at least one valid PRECOMMIT for  $c'$  during attempt  $a'$ , and all other valid PRECOMMITs during attempt  $a'$  must be for the same  $c'$  by 3.5.6). Finally, if  $j$  had an active PRECOMMIT from an attempt  $< a'$ , then it will become inactive once the VOTEFOR with all its dependencies (including the newer PRECOMMIT( $c'$ )) has been delivered to this process  $j$  (cf. 3.5.7), and the process will again vote for the value  $c$  indicated in VOTEFOR. Therefore, all processes from  $I^+$  will vote for the same  $c = c'$  during this attempt, less than  $K - 2\tau$  seconds after the beginning of the attempt (with some probability bounded away from zero).
- If there are no PRECOMMITs yet, then the above reasoning simplifies further: all processes from  $I^+$  that receive this VOTEFOR will immediately vote for the candidate  $c$  suggested by this VOTEFOR.
- In both cases, all processes from  $I^+$  will create a VOTE for the same candidate  $c$  less than  $K - 2\tau$  seconds from the beginning of the attempt,

and this will happen with a positive probability bounded away from zero.

- Finally, all processes from  $I^+$  will receive these VOTES for  $c$  from all processes from  $I^+$ , again less than  $(K - 2\tau) + \tau = K - \tau$  seconds after the beginning of this attempt, i.e., still during the same attempt (even after taking into account the imperfect clock synchronization between processes from  $I^+$ ). This means that they will all create a valid PRECOMMIT for  $c$ , i.e., the protocol will accept  $c$  during this attempt with probability bounded away from zero.
- Since there are infinitely many “very good” attempts, and the probability of successful termination during each such attempt is  $\geq p > 0$  for some fixed value of  $p$ , the protocol will terminate successfully with probability one.

## References

- [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.