Catchain 共識協議：概述

Nikolai Durov
譯者：Dr. Awesome Doge

October 31, 2025

Abstract

本文旨在提供 Catchain 共識協議（Catchain Consensus Protocol）的概述，這是一種專門為 TON 區塊鏈中的區塊生成與驗證所設計的拜占庭容錯（Byzantine Fault Tolerant, BFT）協議 [3]。此協議也可能用於權益證明（proof-of-stake, PoS）區塊鏈中區塊生成以外的其他用途；然而，當前的實作使用了一些僅針對此特定問題有效的最佳化方法。

Contents

1 概述

Catchain 共識協議建立在 TON 網路的覆蓋網路建構協議與覆蓋網路廣播協議之上（[3]）。Catchain 共識協議本身可分解為兩個獨立的協議，一個是較低層級且通用的協議（*Catchain* 協議[1]），另一個是高層級的區塊共識協議（*Block Consensus Protocol, BCP*），後者會使用 Catchain 協議。TON 協議堆疊中的更高層級由區塊生成與區塊驗證層級所佔據；然而，這些層級基本上都在一台（邏輯上的）機器上本地執行，而在新生成的區塊上達成共識的問題則委派給 Catchain 協議層級處理。

以下是 TON 用於區塊生成與分發的協議堆疊近似圖，顯示了 Catchain 共識協議（或更確切地說，其兩個組成協議）的正確位置：

- 頂層：區塊生成與區塊驗證軟體，邏輯上運行在獨立的邏輯機器上，所有輸入由較低層級協議提供，輸出也由較低層級協議處理。此軟體的工作是為區塊鏈（TON 區塊鏈的分片鏈或主鏈；參見 [3] 中關於分片鏈和主鏈的討論）生成新的有效區塊，或檢查由其他人生成的區塊的有效性。

- （*TON*）區塊共識協議：在當前驗證者群組中，針對主鏈或分片鏈達成（拜占庭容錯）共識，決定下一個要接受的區塊。此層級使用區塊生成與驗證軟體的（抽象介面），並建立在較低層級的 Catchain 協議之上。此協議在第 **3** 節中有更詳細的說明。

- *Catchain* 協議：在覆蓋網路中提供安全的持久性廣播（例如，專門用於特定分片鏈或主鏈的區塊生成、驗證與傳播的驗證者任務群組），並偵測部分參與者的「作弊」（違反協議）嘗試。此協議在第 **2** 節中有更詳細的說明。

- （*TON* 網路）覆蓋廣播協議：如 [3] 中所述，用於 TON 網路中覆蓋網路的簡單盡力而為廣播協議。只是將收到的廣播訊息廣播給同一覆蓋網路中尚未收到這些訊息副本的所有鄰居節點，並以最小的努力在短時間內保留未傳遞廣播訊息的副本。

- （*TON* 網路）覆蓋協議：在 ADNL 協議網路內建立覆蓋網路（參見 [3]），管理這些覆蓋網路的鄰居節點列表。覆蓋網路的每個參與者追蹤同一覆蓋網路中的數個鄰居節點，並與它們保持專用的 ADNL

[1]在研究與開發階段初期使用的原始名稱為 *catch-chain* 或 *catchchain*，因為它本質上是一個專門用於捕捉（*catch*）共識協議中所有重要事件的特殊區塊鏈（block*chain*）；在多次使用與書寫這個名稱後，逐漸縮寫為「catchain」。

連線（稱為「通道」），以便能夠以最小的開銷高效地將傳入訊息廣播給所有鄰居節點。

- 抽象資料包網路層（*Abstract Datagram Network Layer, ADNL*）協議：TON 網路的基礎協議，在僅由 256 位元抽象（ADNL）地址識別的網路節點之間傳遞封包（資料包），這些地址實際上是密碼金鑰（或其雜湊值）。

本文旨在僅描述此協議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。

我們在此要指出，本文的作者雖然提供了此協議應如何設計的一般指導方針（例如「讓我們建立一個經 BFT 強化的群組廣播訊息系統，並在此系統之上運行適當調整的簡單兩階段或三階段提交協議」），並參與了協議開發與實作期間的多次討論，但絕不是此協議的唯一設計者，尤其不是其當前實作的唯一設計者。這是多人共同完成的工作。

關於 Catchain 共識協議整體效率的幾點說明。首先，它是一個真正的拜占庭容錯（BFT）協議，意思是即使部分參與者（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，它最終仍能在區塊鏈的下一個有效區塊上達成共識。眾所周知，如果至少三分之一的參與者是惡意的，則不可能達成 BFT 共識（參見 [5]），因此 Catchain 共識協議在這方面已經達到理論上可能的最佳狀態。其次，當 Catchain 共識於 2018 年 12 月首次實作並在分布於全球的多達 300 個節點上進行測試時，它在 300 個節點時達成新區塊共識需要 6 秒，在 100 個節點時需要 4–5 秒（在 10 個節點時需要 3 秒），即使部分節點無法參與或表現出不正確的行為。[2]由於 TON 區塊鏈任務群組預期不會包含超過一百個驗證者（即使總共有一千或一萬個驗證者在運行，也只有其中質押最大的一百個會生成新的主鏈區塊，其他驗證者僅參與新分片鏈區塊的建立，每個分片鏈區塊由 10–30 個驗證者生成與驗證；當然，這裡給出的所有數字都是配置參數（參見 [3] 和 [4]），如有必要可在稍後透過驗證者的共識投票進行調整），這意味著 TON 區塊鏈能夠如最初計劃的那樣，每 4–5 秒生成一次新區塊。這個承諾在幾個月後（2019 年 3 月）TON 區塊鏈測試網路啟動時得到了進一步測試和驗證。因此，我們看到 Catchain 共識協議是不斷增長的實用 BFT 協議家族中的新成員（參見 [2]），儘管它基於略有不同的原則。

[2]當惡意、不參與或非常緩慢的驗證者比例增長到接近三分之一時，協議會展現出優雅降級（graceful degradation），區塊共識時間增長非常緩慢——例如，最多增加半秒——直到接近三分之一的臨界值。

2 Catchain 協議

我們在概述中已經解釋過（參見 **1**），TON 區塊鏈用於在新區塊鏈區塊上達成共識的 BFT 共識協議由兩個協議組成。我們在此提供 *Catchain* 協議的簡要描述，它是這兩個協議中較低層級的協議，可能用於區塊 BFT 共識以外的其他用途。Catchain 協議的原始碼位於原始碼樹的 `catchain` 子目錄中。

2.1. 運行 Catchain 協議的先決條件. 運行（一個實例的）Catchain 協議的主要先決條件是所有參與（或允許參與）此特定協議實例的節點的有序列表。此列表包含所有參與節點的公開金鑰和 ADNL 地址。在建立 Catchain 協議實例時，必須從外部提供此列表。

2.2. 參與區塊共識協議的節點. 為了完成為 TON 區塊鏈的其中一條區塊鏈（即主鏈或其中一條活躍的分片鏈）建立新區塊的特定任務，會建立一個由數個驗證者組成的特殊任務群組。此任務群組的成員列表既用於在 ADNL 內建立私有覆蓋網路（這意味著只有在建立時明確列出的節點才能加入此覆蓋網路），也用於運行對應的 Catchain 協議實例。

此成員列表的建構是整體協議堆疊較高層級（區塊建立與驗證軟體）的責任，因此不是本文的主題（[4] 將是更合適的參考資料）。目前只需知道此列表是當前（最新）主鏈狀態的確定性函數（尤其是配置參數的當前值，例如所有被選出用於建立新區塊的驗證者的活躍列表及其各自的權重）。由於列表是確定性計算的，所有驗證者都會計算出相同的列表，特別是每個驗證者無需任何進一步的網路通訊或協商即可知道自己參與了哪些任務群組（即 Catchain 協議的實例）。[3]

2.2.1. Catchain 會預先建立. 事實上，不僅會計算上述列表的當前值，也會計算其緊接著的（未來）值，因此 Catchain 通常會預先建立。這樣一來，當新的驗證者任務群組實例必須建立第一個區塊時，它已經就緒。

2.3. 創世區塊與 catchain 識別符. 一個 *catchain*（即 Catchain 協議的一個實例）由其創世區塊或創世訊息來表徵。這是一個簡單的資料結構，包含一些魔術數字、catchain 的目的（例如，將生成區塊的分片鏈識別符，以及所謂的 *catchain* 序號，也從主鏈配置中取得，用於區分生成「相同」分片鏈但可能有不同參與驗證者的後續 catchain 實例），以及最重要的，所有參與節點的列表（它們的 ADNL 地址和 Ed25519 公開金鑰，如 **2.1** 中所

[3]如果某些驗證者擁有過時的主鏈狀態，它們可能無法計算出正確的任務群組列表並參與對應的 catchain；在這方面，它們被視為惡意或故障，只要以這種方式失敗的驗證者少於所有驗證者的三分之一，就不會影響 BFT 協議的整體有效性。

述）。Catchain 協議本身只使用此列表和整體資料結構的 sha256 雜湊；此雜湊用作 catchain 的內部識別符，即此特定 Catchain 協議實例的識別符。

2.3.1. 創世區塊的分發. 請注意，創世區塊不會在參與節點之間分發；而是由每個參與節點獨立計算，如 **2.2** 中所述。由於創世區塊的雜湊被用作 catchain 識別符（即特定 Catchain 協議實例的識別符；參見 **2.3**），如果某個節點（意外或故意）計算出不同的創世區塊，它將實際上被鎖定在「正確」協議實例之外無法參與。

2.3.2. 參與 catchain 的節點列表. 請注意，參與 catchain 的節點（有序）列表固定在創世區塊中，因此所有參與者都知道它，並且只要 sha256 沒有（已知的）碰撞，它就由創世區塊的雜湊（即 catchain 識別符）明確決定。因此，在下文討論一個特定 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（它們的真實身份可以使用範圍 $1 \dots N$ 中的此索引在參與者列表中查找）。所有參與者的集合將表示為 I；我們假設 $I = \{1 \dots N\}$。

2.4. Catchain 中的訊息。Catchain 作為處理程序群組. 一個觀點是，catchain 是一個（分散式）處理程序群組，由 N 個已知且固定的（通訊）處理程序（或在前述術語中的節點）組成，這些處理程序生成廣播訊息，最終會廣播給處理程序群組的所有成員。所有處理程序的集合表示為 I；我們通常假設 $I = \{1 \dots N\}$。每個處理程序生成的廣播從一開始編號，因此處理程序 i 的第 n 次廣播將獲得序號或高度 n；每個廣播應該由發起處理程序的身份或索引 i 及其高度 n 唯一決定，因此我們可以將配對 (i, n) 視為處理程序群組內廣播訊息的自然識別符。[4]預期由同一處理程序 i 生成的廣播將以它們被建立的完全相同順序傳遞給每個其他處理程序，即按其高度的遞增順序。在這方面，catchain 與 [1] 或 [7] 意義上的處理程序群組非常相似。主要區別在於 catchain 是處理程序群組的「強化」版本，能夠容忍部分參與者可能的拜占庭（任意惡意）行為。

2.4.1. 訊息上的依賴關係. 可以在處理程序群組中廣播的所有訊息上引入一個依賴關係。此關係必須是嚴格偏序 $\prec$，具有性質 $m_{i,k} \prec m_{i,k+1}$，其中 $m_{i,k}$ 表示索引為 i 的群組成員處理程序廣播的第 k 個訊息。$m \prec m'$ 的意義是m' 依賴於 m，因此（廣播）訊息 m' 只能在 m 已被處理之後才能被處理（由處理程序群組的成員）。例如，如果訊息 m' 代表群組成員對另一個訊息 m 的反應，那麼自然設定 $m \prec m'$。如果處理程序群組的成員在所有其依賴項（即訊息 $m \prec m'$）被處理（或傳遞給較高層級協議）之前接收到訊息 m'，則其處理（或傳遞）會延遲，直到所有其依賴項都被傳遞。

[4]在 catchain 的拜占庭環境中，這並非在所有情況下都成立。

我們將依賴關係定義為嚴格偏序，因此它必須是遞移的（$m'' \prec m'$ 和 $m' \prec m$ 隱含 $m'' \prec m$）、反對稱的（對於任意兩個訊息 m 和 m'，$m' \prec m$ 和 $m \prec m'$ 最多只有一個成立）和反自反的（$m \prec m$ 永遠不成立）。如果我們有一個較小的「基本依賴」集合 $m' \to m$，我們可以構造其遞移閉包 $\to^+$ 並設定 $\prec := \to^+$。唯一的其他要求是發送者的每次廣播都依賴於同一發送者的所有先前廣播。嚴格來說不必假設這一點；然而，這個假設相當自然，並且大大簡化了處理程序群組內訊息系統的設計，因此 Catchain 協議採用了這個假設。

2.4.2. 訊息的依賴集或錐體. 設 m 為上述處理程序群組內的（廣播）訊息。我們稱集合 $D_m := \{m' : m' \prec m\}$ 為訊息 m 的依賴集或依賴錐體。換句話說，D_m 是由 m 在所有訊息的部分有序有限集合中生成的主理想。它恰好是在 m 被傳遞之前必須傳遞的所有訊息的集合。

2.4.3. 訊息的擴展依賴錐體. 我們也定義 D_m^+，即m 的擴展依賴錐體，為 $D_m^+ := D_m \cup \{m\}$。

2.4.4. 錐體，或關於 $\prec$ 的理想. 更一般地，我們稱訊息的子集 D 為錐體，如果它是關於依賴關係 $\prec$ 的理想，即如果 $m \in D$ 和 $m' \prec m$ 隱含 $m' \in D$。當然，任何訊息 m 的依賴錐體 D_m 和擴展依賴錐體 D_m^+ 都是錐體（因為部分有序集合中的任何主理想都是理想）。

2.4.5. 藉助向量時間識別錐體. 回想一下，我們假設任何訊息都依賴於同一發送者的所有先前訊息，即對於任何 $i \in I$ 和任何 $s > 0$（使得 $m_{i,s+1}$ 存在），有 $m_{i,s} \prec m_{i,s+1}$。這意味著任何錐體 D 完全由 N 個由 $i \in I$ 索引的值 $\mathrm{VT}(D)_i$ 表徵：

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

（如果沒有訊息 $m_{i,s}$ 在 D 中，我們設定 $\mathrm{VT}(D)_i := 0$）。實際上，顯然

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

我們稱具有非負分量 $\mathrm{VT}(D)_i$ 的向量 $\mathrm{VT}(D) = (\mathrm{VT}(D)_i)_{i \in I} \in \mathbb{N}_0^I$ 為對應於錐體 D 的向量時間或向量時間戳（參見 [1] 或 [7] 以獲得關於向量時間的更詳細討論）。

2.4.6. 向量時間戳上的偏序. 我們在所有可能的向量時間集合 $\mathbb{N}_0^I$ 上引入偏序 $\le$，這是 $\mathbb{N}_0$ 上通常順序的乘積：

$$\boldsymbol{x} = (x_i)_{i \in I} \le \boldsymbol{y} = (y_i)_{i \in I} \quad \text{若且唯若} \quad x_i \le y_i \quad \text{對所有 } i \in I \text{ 成立} \qquad (3)$$

立即可得 $D \subset D'$ 若且唯若 $\mathrm{VT}(D) \le \mathrm{VT}(D')$；因此，VT 是所有錐體集合（包含在所有訊息集合中）到 $\mathbb{N}_0^I$ 的嚴格保序嵌入。

2.4.7. 訊息 m 的向量時間戳 $\mathbf{Vt}(m)$. 給定任何訊息 m，我們將其向量時間戳 $\mathrm{VT}(m)$ 定義為 $\mathrm{VT}(D_m)$。換句話說，訊息 m 只能在處理程序 j 生成的前 $\mathrm{VT}(m)_j$ 個訊息被傳遞之後才能被傳遞，這對所有 $j \in I$ 都成立。

如果 i 是訊息 m 的發送者，s 是訊息 m 的高度，使得 $m = m_{i,s}$，則 $\mathrm{VT}(m)_i = s - 1$。我們可以定義訊息 m 的調整向量時間戳 $\mathrm{VT}^+(m)$，設定 $VT^+(m)_j = VT(m)_j$（對 $j \neq i$），$VT^+(m)_i = \mathrm{VT}(m)_i + 1 = s$。或者，$\mathrm{VT}^+(m) = \mathrm{VT}(D_m^+)$，其中 $D_m^+ := D_m \cup \{m\}$ 是m 的擴展依賴錐體（參見 **2.4.3**）。

請注意，$m' \preceq m$ 若且唯若 $D_{m'}^+ \subset D_m^+$ 若且唯若 $\mathrm{VT}^+(m') \leq \mathrm{VT}^+(m)$（在 $\mathbb{N}_0^I$ 中），其中 $m' \preceq m$ 表示「$m' \prec m$ 或 $m' = m$」。類似地，$m' \prec m$ 若且唯若 $D_{m'}^+ \subset D_m$ 若且唯若 $\mathrm{VT}^+(m') \leq \mathrm{VT}(m)$。換句話說，（部分或所有）訊息上的依賴關係 $\prec$ 完全由這些訊息的調整向量時間戳決定。

2.4.8. 使用向量時間戳正確傳遞廣播訊息. 向量時間戳可用於（在非拜占庭環境中）正確傳遞處理程序群組中的廣播訊息。[5]也就是說，假設每個廣播訊息 $m = m_{i,s}$ 都包含其發送者的索引 i 和此訊息的向量時間戳 $\mathrm{VT}(m)$。那麼每個接收者 j 都知道該訊息是否可以被傳遞。為此，j 追蹤迄今為止傳遞的所有訊息的錐體 C_j，例如透過維護一個等於 $\mathrm{VT}(C_j)$ 的當前時間戳 $\mathrm{VT}(j)$。換句話說，$\mathrm{VT}(j)_k$ 是 j 迄今為止處理的發送者 k 的訊息計數。如果 $\mathrm{VT}(m) \leq \mathrm{VT}(j)$，則訊息 m 立即被傳遞，且 $\mathrm{VT}(j)$ 之後更新為 $\sup(\mathrm{VT}(j), \mathrm{VT}^+(m))$；這等價於將 $\mathrm{VT}(j)_i$ 增加一，其中 i 是訊息 m 的原始發送者。如果不滿足此條件，則 m 可能被放入等待佇列，直到 $\mathrm{VT}(j)$ 變得足夠大。j 可以構造某個已接收但未傳遞訊息 m 的 $\mathrm{VT}(m)$ 中隱含提及的訊息索引 (i', s') 列表，並從其了解到 m 和 $\mathrm{VT}(m)$ 的鄰居節點請求具有這些索引的訊息，而不是被動等待所需的廣播；另一種策略（Catchain 協議的當前實作實際採用的）是不時地從隨機選擇的鄰居節點請求這些訊息。後一種策略更簡單，因為它不需要記住所有已接收訊息的直接來源（無論如何這些來源可能變得不可用）。

2.5. Catchain 中的訊息結構。Catchain 作為多重區塊鏈. 由於需要支援 BFT 協議，catchain 中的訊息結構比上述描述稍微複雜一些。特別是，在拜占庭環境中，向量時間戳是不夠的。它們必須由基於依賴錐體的最大元素的描述來補充（這種描述通常僅在非拜占庭環境中，當處理程序群組非常大以致向量時間戳大小變得過大時才使用）。

2.5.1. 藉由最大元素描述錐體. 描述訊息錐體 D 的另一種方法（與使用向量時間戳相對）是列出其所有最大元素 $\mathrm{Max}(D)$，即元素 $m \in D$，使得對

[5]我們假設處理程序群組中的所有廣播訊息都是 [1] 術語中的「因果廣播」或「cbcast」，因為我們只需要 cbcast 來實作 Catchain 協議和 Catchain 共識。

於任何 $m' \in D$ 都不成立 $m \prec m'$。當然，為了使這種表示法實用，需要一種合適的方式來引用訊息而不完全包含它們。

2.5.2. Catchain 內的訊息識別符. Catchain 協議使用（適當序列化的）訊息的*SHA256* 雜湊作為其唯一識別符。如果我們假設 SHA256 沒有碰撞（在合理時間內，例如多項式時間內可計算），那麼訊息 m 在處理程序群組內完全由其雜湊 $\text{SHA256}(m)$ 識別。

2.5.3. 訊息標頭. catchain 內（即 Catchain 協議的一個實例）訊息 $m = m_{i,s}$ 的標頭總是包含其發送者的索引 i、高度 s、catchain 識別符（即創世訊息的雜湊，參見 **2.3**）以及 m 的依賴錐體的最大元素的雜湊集合，即集合 $\{\text{SHA256}(m') : m' \in \text{Max}(D_m)\}$。特別是，由於當 $s > 1$ 時 $m_{i,s-1} \in \text{Max}(D_m)$，同一發送者的前一個訊息的雜湊 $\text{SHA256}(m_{i,s-1})$ 總是被包含；出於性能原因，訊息標頭中有一個單獨的欄位包含 $\text{SHA256}(m_{i,s-1})$。如果 $s = 1$，則沒有前一個訊息，因此使用創世訊息的雜湊（即 catchain 識別符，參見 **2.3**）代替。

向量時間戳 $\text{VT}(m)$ 不包含在訊息標頭中；然而，標頭隱含地決定了 $\text{VT}(m)$，因為

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

請注意，訊息標頭是訊息的一部分，特別是訊息的雜湊（即訊息識別符）依賴於標頭中列出的所有資料。因此，我們假設訊息識別符隱含地決定了對應訊息的所有依賴項（如果 SHA256 沒有已知的碰撞）。

2.5.4. 訊息簽章. 除此之外，catchain 中的每個訊息都由其建立者簽章。由於 catchain 中參與節點（處理程序）的列表是預先已知的，並且此列表包含所有處理程序的公開金鑰，因此接收處理程序可以在接收到訊息後立即檢查這些訊息簽章。如果簽章無效，訊息將被丟棄而不進行任何進一步處理。

2.5.5. 訊息加密. catchain 中的所有訊息在從一個節點傳輸到 catchain 底層私有覆蓋網路中的鄰居節點之前也會被加密。然而，這種加密是由較低層級的網路協議（例如 ADNL）執行的，與此處的討論無關。我們想提及的是，此處的正確加密之所以可能，僅僅是因為參與處理程序的列表不僅包含所有處理程序的公開金鑰，還包含它們的 ADNL 地址（實際上是用於網路傳輸的公開加密金鑰）。

請注意，即使沒有加密，這也不會違反協議的 BFT 屬性，因為由於簽章的存在，無法偽造來自其他發送者的訊息。然而，這可能導致資訊洩漏給外部觀察者，這通常是不希望出現的。

2.5.6. 另一種觀點：catchain 作為多重區塊鏈. 請注意，catchain 中由同一發送者 i 建立的所有訊息原來具有簡單的「區塊鏈結構」，因為 $m_{i,s+1}$ 的標頭包含發送者 i 的前一個訊息的雜湊 $\text{SHA256}(m_{i,s})$（在來自 $\text{Max}(D_{m_{i,s+1}})$ 的訊息的其他雜湊中）。這樣，每個處理程序 i 生成一個由其訊息組成的簡單區塊鏈，這個區塊鏈的每個「區塊」對應一個訊息並透過其雜湊引用前一個區塊，有時還透過在其區塊中提及這些區塊的雜湊來包含對其他處理程序的區塊（即訊息）的引用。每個區塊都由其建立者簽章。所得結構與 [3, 5] 中考慮的「非同步支付通道」非常相似，但有 N 個參與者而不是 2 個。

2.6. Catchain 中的訊息傳播. 現在我們準備描述 catchain 中的訊息傳播。也就是說：

- （較低層級的）覆蓋網路協議維護 catchain 底層私有覆蓋網路中的鄰居節點列表，並為每個鄰居節點提供 ADNL 通道。此私有覆蓋網路與 catchain 具有相同的成員（處理程序、節點）列表，每個節點的鄰居節點在所有參與節點的集合上形成一個（有向）子圖。這個（本質上是隨機的）子圖以非常接近一的機率是強連通的。

- 每個處理程序不時地生成一些新訊息（根據較高層級協議的需要）。這些訊息按照 **2.5.3** 中概述的方式增強為 catchain 訊息標頭，簽章，並使用覆蓋協議建立的 ADNL 通道傳播給所有已知的鄰居節點。

- 與通常的簡單覆蓋廣播協議相反，從鄰居節點接收的訊息不會立即重新廣播給尚不知道擁有其副本的所有其他鄰居節點。相反，首先檢查簽章，無效的訊息被丟棄。然後訊息要麼被傳遞（如果其所有依賴訊息都已被傳遞），要麼被放入等待佇列。在後一種情況下，從發送此訊息的鄰居節點拉取其標頭中提及的所有所需訊息（即集合 $\text{Max}(D_m)$）（除此之外，還會不時嘗試從隨機鄰居節點下載這些缺失的訊息）。如有必要，此過程會遞迴重複，直到某些訊息可以被傳遞。一旦訊息準備好進行本地傳遞（即其所有依賴項都已存在），它也會重新廣播給覆蓋網路中的所有鄰居節點。

- 除了上述遞迴「拉取」機制外，還使用了更快的基於向量時間戳的機制，以便可以透過發送者和高度（從接收到的訊息的向量時間戳中得知）從鄰居節點查詢訊息。也就是說，每個處理程序不時地向隨機選擇的鄰居節點發送包含當前向量時間戳的特殊查詢。此對等查詢導致其接收者傳回發送者未知的所有或部分訊息（根據它們的向量時間戳判斷）。

- 一旦偵測到「分叉」，即從鄰居節點得知具有相同發送者 i 和高度 s 但具有不同雜湊的第二個訊息（例如，在快速或慢速「拉取」過程中），就可以對源自某些發送者的訊息停用這種更快的基於向量時間戳的機制。一旦偵測到由 i 建立的分叉，所有後續向量時間戳的對應分量 VT_i 都會設定為特殊值 ∞，以指示比較這些分量的值不再有意義。

- 當訊息被傳遞（給較高層級協議）時，此訊息被添加到當前處理程序的已處理訊息錐體 C 中（並相應地更新當前向量時間戳），當前處理程序生成的所有後續訊息將被假定依賴於迄今為止傳遞的所有訊息（即使從較高層級協議的角度來看，這在邏輯上不是必需的）。

- 如果已處理訊息錐體的最大元素集合 $\mathrm{Max}(C)$ 變得太大（包含的元素多於 catchain 的創世訊息預先固定的某個數量），則 Catchain 協議會要求較高層級協議生成新訊息（如果沒有可用的有用有效負載，則為空訊息）。在生成此新訊息（並立即傳遞給當前處理程序）之後，C 被更新，$\mathrm{Max}(C)$ 僅包含一個元素（新訊息）。這樣，$\mathrm{Max}(C)$ 的大小以及訊息標頭的大小總是保持有界。

- 一旦訊息 m 被傳遞且集合 C 被修改以包含此訊息，就會設定計時器，在一些小延遲後要求較高層級協議建立新訊息（如有必要則為空訊息），以便此新訊息 m^* 引用新的 C，類似於前一項中描述的過程。此新訊息 m^* 被推送給所有鄰居節點；由於其標頭包含新 C 的 $\mathrm{Max}(C)$，且 $m \in C$，鄰居節點不僅了解新生成的訊息 m^*，還了解原始接收到的訊息 m。如果某些鄰居節點還沒有 m 的副本，它們會要求一個（從當前處理程序或其他處理程序）。

- catchain 中接收和建立的所有（廣播）訊息都儲存到特殊的本地資料庫中。這對於新建立的訊息尤其重要（參見 **3.3.2**）：如果訊息被建立並發送給鄰居節點，但在建立處理程序崩潰並重新啟動之前未儲存到資料庫（並刷新到磁碟），則重新啟動後可以建立具有相同發送者和高度的另一個訊息，從而實際上導致非自願的「分叉」。

2.7. 分叉及其預防. 可以看到，上述 catchain 的多重區塊鏈結構（透過雜湊引用其他區塊並帶有簽章）幾乎沒有為建立在 catchain 之上的共識協議（即使用 catchain 作為在處理程序群組內廣播訊息的手段）中的「作弊」留下任何可能性。唯一不會立即被偵測到的可能性在於建立同一訊息 $m_{i,s}$ 的兩個（或更多）不同版本（例如 $m'_{i,s}$ 和 $m''_{i,s}$），並將此訊息的一個版本 $m'_{i,s}$ 發送給某些對等節點，將不同版本 $m''_{i,s}$ 發送給其他對等節點。如果 s 是最

小的（對於固定的 i），則這對應於區塊鏈術語中的分叉：相同前一個區塊 $m_{i,s-1}$ 的兩個不同的下一個區塊 $m'_{i,s}$ 和 $m''_{i,s}$。

因此，Catchain 協議會盡快偵測分叉並防止其傳播。

2.7.1. 分叉的偵測. 分叉的偵測很簡單：如果有兩個不同的區塊 $m'_{i,s}$ 和 $m''_{i,s}$ 具有相同的建立者 $i \in I$ 和相同的高度 $s \geq 1$，並且具有 i 的有效簽章，則這是一個分叉。

2.7.2. 分叉證明. Catchain 協議中的區塊簽章以這樣一種方式建立，使得建立分叉證明（即處理程序 i 故意建立分叉的證明）特別簡單，因為實際簽章的是一個非常小的結構（包含魔術數字、i 和 s 的值以及訊息其餘部分的雜湊）的雜湊。因此，分叉證明中只需要兩個這樣的小結構和兩個簽章。

2.7.3. 建立分叉的外部懲罰. 請注意，在權益證明區塊鏈生成環境中，可以對建立 catchain 分叉使用外部懲罰。也就是說，分叉證明可以提交給特殊的智慧合約（例如 TON 區塊鏈的選舉智慧合約），自動檢查，並且可以沒收違規方的部分或全部質押。

2.7.4. 分叉的內部處理. 一旦（由 i 建立的）分叉被偵測到（由另一個處理程序 j），即 j 得知由 i 建立且具有相同高度 s 的兩個不同訊息 $m_{i,s}$ 和 $m'_{i,s}$（通常這發生在遞迴下載某些其他訊息的依賴項時），j 開始忽略 i 及其所有後續訊息。它們不會被接受，也不會進一步廣播。然而，如果在偵測到分叉之前引用了由 i 建立的這些訊息，則由處理程序建立的訊息（區塊）中引用的由 i 在分叉偵測之前建立的訊息仍可能被下載。

2.7.5. 接受來自「壞」處理程序的訊息是壞的. 此外，如果處理程序 i 得知處理程序 j 建立了分叉，則 i 透過建立包含對應分叉證明（參見 **2.7.2**）的新服務廣播訊息向其鄰居節點顯示這一點。之後，j 的這個訊息和所有後續訊息都不能直接依賴於已知「壞」生產者 i 的任何訊息（但如果在建立引用訊息時 k 不知道 i 的分叉，它們仍然可以引用來自另一方 k 的訊息，而該訊息直接或間接引用 i 的訊息）。如果 j 違反此限制並建立具有這種無效引用的訊息，這些訊息將被群組中的所有誠實處理程序丟棄。

2.7.6. 「壞」群組成員的集合是內在狀態的一部分. 每個處理程序 i 保留其自己的群組中已知「壞」處理程序集合的副本，即那些已建立至少一個分叉或違反 **2.7.5** 的處理程序。一旦 i 得知 j 建立的分叉（或 j 對 **2.7.5** 的違反），此集合就會透過將 j 添加到其中而更新；之後，會呼叫較高層級協議提供的回呼。當新的廣播訊息到達時使用此集合：如果發送者是壞的，則訊息被忽略並丟棄。

3 區塊共識協議

本節將解釋 TON 區塊共識協議（參見 **1**）的基本運作方式，該協議建立在通用 Catchain 協議（參見 **2**）之上，提供用於生成和驗證 TON 區塊鏈新區塊的 BFT 協議。TON 區塊共識協議的原始碼位於原始碼樹的 `validator-session` 子目錄中。

3.1. 區塊共識協議的內部狀態. 較高層級的區塊共識協議向 catchain 引入了一個新概念：區塊共識協議（BCP）的內部狀態，有時也（不太正確地）稱為「catchain 的內部狀態」或簡稱 *catchain* 狀態。也就是說，在 Catchain 協議將訊息子集（實際上總是依賴錐體）C_i 傳遞給較高層級協議（即在此情況下為區塊共識協議）之後，每個處理程序 $i \in I$ 都有一個明確定義的內部狀態 σ_{C_i}。此外，此狀態 $\sigma_{C_i} = \sigma(C_i)$ 僅依賴於錐體 C_i，而不依賴於處理程序 $i \in I$ 的身份，並且可以為任何依賴錐體 S 定義（不一定是某個處理程序 i 在某個時刻的已傳遞訊息錐體 C_i）。

3.1.1. 內部狀態的抽象結構. 我們從 BCP 採用的內部狀態的抽象結構開始；稍後將提供更具體的細節。

3.1.2. 更新內部狀態. Catchain 協議對內部狀態一無所知；它只是在每次傳遞訊息 m 時呼叫較高層級協議（即 BCP）提供的適當回呼。較高層級協議的工作是從先前計算的狀態 σ_S 和訊息 m 開始計算新狀態 $\sigma_{S'}$，其中 $S' = S \cup \{m\}$（且必然 $S \supset D_m$，否則此時無法傳遞 m）。

3.1.3. 更新內部狀態的遞迴公式. 計算所有錐體 S 的 σ_S 的抽象設定由三個組成部分構成：

- 初始狀態的值 $\sigma_\emptyset$（此值實際上依賴於 catchain 的創世區塊；我們在此忽略此依賴性，因為目前只考慮一個 catchain）。
- 函數 f，從前一個狀態 σ_{D_m} 和新傳遞的訊息 m 計算狀態 $\sigma_{D_m^+}$：

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

  其中 D_m 是訊息 m 的依賴錐體，$D_m^+ = D_m \cup \{m\}$ 是其擴展依賴錐體（參見 **2.4.3**）。在大多數情況下，f 實際上會滿足更強的條件

$$\sigma_{S\cup\{m\}} = f(\sigma_S, m) \quad \text{若 } S \text{ 和 } S\cup\{m\} \text{ 是錐體且 } m \notin S \tag{6}$$

  然而，更新演算法不需要此更強條件。

- 「合併函數」g，從 σ_S 和 σ_T 計算 $\sigma_{S\cup T}$：

$$\sigma_{S\cup T} = g(\sigma_S, \sigma_T) \quad \text{對任意錐體 } S \text{ 和 } T \tag{7}$$

（兩個錐體的聯集總是錐體）。此函數 σ 由更新演算法僅在特定情況下應用，即 $T = D_m^+$ 且 $m \notin S$。

3.1.4. g 的可交換性與結合性. 請注意，(7)（對於任意錐體 S 和 T）隱含 g 的結合性和可交換性，至少當 g 應用於可能的狀態（某個錐體 S 的形式為 σ_S 的值）時。在這方面，g 在集合 $\Sigma = \{\sigma_S : S \text{ 是錐體 }\}$ 上定義了可交換單元結構。通常 g 在更大的狀態類值集合 $\tilde{\Sigma}$ 上定義或部分定義，並且它可能在這個更大的集合 $\tilde{\Sigma}$ 上是可交換和結合的，即對於 x, y, $z \in \tilde{\Sigma}$（當等式兩邊都定義時），有 $g(x, y) = g(y, x)$ 和 $g(x, g(y, z)) = g(g(x, y), z)$，並以 $\sigma_\emptyset$ 作為單位元，即對於 $x \in \tilde{S}$（在相同條件下），有 $g(x, \sigma_\emptyset) = x = g(\sigma_\emptyset, x)$。然而，此性質雖然對於共識演算法的形式分析有用，但狀態更新演算法並不嚴格要求，因為此演算法以確定性方式使用 g 來計算 σ_S。

3.1.5. f 的可交換性. 請注意，如果 f 滿足更強條件 (6)，則它也必須表現出可交換性性質

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

當 S 是錐體且 m 和 m' 是兩個訊息，滿足 $D_m \subset S$、$D_{m'} \subset S$、$m \notin S$ 和 $m' \notin S$ 時，因為在這種情況下 $S \cup \{m\}$、$S \cup \{m'\}$ 和 $S \cup \{m, m'\}$ 也是錐體，並且 (6) 隱含 (8) 的兩邊都等於 $\sigma_{S\cup\{m,m'\}}$。類似於 **3.1.4**，f 通常在更大的狀態類值集合 $\tilde{\Sigma}$ 和訊息類值集合的乘積上定義或部分定義；它可能在這個更大的集合上表現出「可交換性」性質 (8)，也可能不表現。如果表現出此性質，這可能對依賴於 σ_S 的演算法的形式分析有用，但此性質並非嚴格必需。

3.1.6. 狀態更新演算法. catchain（實際上是較高層級的 BCP）採用的狀態更新演算法（由每個處理程序 i 獨立執行）使用 $\sigma_\emptyset$、f 和 g 如下：

- 演算法追蹤迄今為止傳遞的所有訊息 m 的所有 $\sigma_{D_m^+}$。
- 演算法追蹤 σ_{C_i}，其中 C_i 是當前依賴錐體，即傳遞（給當前處理程序 i）的所有訊息 m 的集合。σ_{C_i} 的初始值是 $\sigma_\emptyset$。
- 當傳遞新訊息 m 時，透過重複應用 g 計算 σ_{D^m} 的值，因為 $D_m = \bigcup_{m'\in D_m} D_{m'}^+ = \bigcup_{m'\in \mathrm{Max}(D_m)} D_{m'}^+$；因此，如果 $\mathrm{Max}(D_m) = \{m'_1, \ldots, m'_k\}$，則

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

集合 $\mathrm{Max}(D_m)$ 以某個固定順序 m'_1, ..., m'_k 明確列在訊息 m 的標頭中；上述公式相對於此順序應用（因此 D_m 的計算是確定性的）。此列表中的第一個元素總是 m 的發送者的前一個訊息，即如果 $m = m_{i,s+1}$，則 $m'_1 = m_{i,s}$。

- 之後，透過應用 f 計算 $\sigma_{D_m^+}$ 的值：$\sigma_{D_m^+} = f(\sigma_{D_m}, m)$。此值被記住以供將來使用。
- 最後，當新訊息 m 傳遞給當前處理程序 i 時，從而將 C_i 更新為 $C'_i := C_i \cup \{m\}$，演算法使用計算的值 $\sigma_{D_m^+}$ 更新當前狀態

$$\sigma_{C'_i} = g(\sigma_{C_i}, \sigma_{D_m^+}) \tag{10}$$

然而，此狀態是「虛擬的」，因為它稍後可能會稍微改變（特別是如果 g 不可交換）。儘管如此，較高層級演算法（BCP）使用它來做出一些重要決策。

- 一旦生成新訊息 m 並在本地傳遞，使得 C_i 變為等於 D_m^+，則先前計算的 σ_{C_i} 值將被丟棄並替換為根據上述一般演算法計算的 $\sigma_{D_m^+}$。如果 g 不可交換或不結合（例如，$g(x, y)$ 和 $g(y, x)$ 可能是相同狀態的不同但等價的表示），則這可能導致處理程序 i 的當前「虛擬」狀態稍微改變。
- 如果較低層級（catchain）協議向較高層級協議報告某個處理程序 $j \notin i$ 是「壞的」（即發現 j 已建立分叉，參見 **2.7.6**，或故意認可另一個處理程序的分叉，參見 **2.7.5**），則使用新集合 $C'_i = \bigcup_{m \in C_i,\, m \text{ 由「好」處理程序 } k \text{ 建立}} D_m^+$ 和應用於 $\sigma_{D_m^+}$ 集合的「合併」函數 g 從頭重新計算當前（虛擬）狀態 σ_{C_i}，其中 m 遍歷已知為好的處理程序的最後訊息集合（或此集合的最大元素集合）。下一個建立的出站訊息將僅依賴於來自 C'_i 的訊息。

3.1.7. 需要知道其他處理程序的內部狀態. 公式 (9) 隱含處理程序 i 也必須追蹤所有訊息 m 的 $\sigma_{D_m^+}$，無論是否由此處理程序建立。然而，這是可能的，因為這些內部狀態也是透過更新演算法的適當應用來計算的。因此，BCP 也會計算並記住所有 $\sigma_{D_m^+}$。

3.1.8. 函數 f 就足夠了. 請注意，更新演算法僅應用 g 來計算 $\sigma_{S \cup D_m^+} = g(\sigma_S, \sigma_{D_m^+})$，當 S 是包含 D_m 但不包含 m 的錐體時。因此，g 的每次實際應用都可以替換為滿足擴展性質 (6) 的 f 的應用：

$$\sigma_{S \cup D_m^+} = g(\sigma_S, \sigma_{D_m^+}) = f(\sigma_S, m) \tag{11}$$

然而，更新演算法不使用此「最佳化」，因為它會停用下面 **3.2.4** 和 **3.2.5** 中描述的更重要的最佳化。

3.2. 內部狀態的結構. 內部狀態的結構經過最佳化，使得 (5) 的轉換函數 f 和 (7) 的合併函數 g 盡可能高效地可計算，最好不需要潛在無界的遞迴（只需要一些迴圈）。這促使在內部狀態中包含額外的組成部分（即使這些組成部分可以從內部狀態的其餘部分計算出來），這些組成部分也必須被儲存和更新。包含額外組成部分的這個過程類似於使用動態規劃解決問題時採用的過程，或類似於透過數學（或結構）歸納法證明陳述時使用的過程。

3.2.1. 內部狀態是抽象代數資料型別值的表示. 內部狀態的內部表示本質上是一個（有向）樹（或更確切地說是有向無環圖）或節點集合；每個節點包含一些直接（通常是整數）值和指向其他（先前建構的）節點的若干指標。如有必要，在節點開頭添加額外的建構子標籤（一個小整數）以區分幾種可能性。此結構與函數式程式語言（例如 Haskell）中用於表示抽象代數資料型別值的結構非常相似。

3.2.2. 內部狀態是持久的. 內部狀態是持久的，意思是在 catchain 活躍時，用於分配作為內部狀態一部分的節點的記憶體永遠不會被釋放。此外，catchain 的內部狀態實際上是在一個巨大的連續記憶體緩衝區內分配的，新節點總是透過推進指標在此緩衝區已使用部分的末端分配。這樣，從此緩衝區內的節點到其他節點的引用可以透過從緩衝區開頭的整數偏移量來表示。每個內部狀態都由指向此緩衝區內其根節點的指標表示；此指標也可以透過從緩衝區開頭的整數偏移量來表示。

3.2.3. Catchain 的內部狀態被刷新到僅附加檔案. 上述用於儲存 catchain 內部狀態的緩衝區結構的結果是，它僅透過在其末端附加一些新資料來更新。這意味著 catchain 的內部狀態（或更確切地說，包含所有所需內部狀態的緩衝區）可以刷新到僅附加檔案，並在重新啟動後輕鬆恢復。在重新啟動之前需要儲存的唯一其他資料是 catchain 當前狀態的偏移量（從緩衝區的開頭，即此檔案的開頭）。簡單的鍵值資料庫可用於此目的。

3.2.4. 在不同狀態之間共享資料. 原來表示新狀態 $\sigma_{S\cup\{m\}} = f(\sigma_S, m)$ 的樹（或更確切地說是 dag）與前一個狀態 σ_S 共享大型子樹，類似地，$\sigma_{S\cup T} = g(\sigma_S, \sigma_T)$ 與 σ_S 和 σ_T 共享大型子樹。BCP 中用於表示狀態的持久結構使得可以重複使用緩衝區內的相同指標來表示這些共享資料結構，而不是複製它們。

3.2.5. 記憶化節點. 計算新狀態（即函數 f 的值）時採用的另一種技術是記憶化新節點，這也是從函數式程式語言中借鑑而來的。也就是說，每當建

構新節點時（在包含特定 catchain 所有狀態的巨大緩衝區內），就會計算其雜湊，並使用簡單的雜湊表查找具有相同雜湊的最新節點。如果找到具有此雜湊的節點，並且它具有相同的內容，則丟棄新建構的節點，並返回對具有相同內容的舊節點的引用。另一方面，如果找不到新節點的副本，則更新雜湊表，推進緩衝區末端（分配）指標，並將新節點的指標返回給呼叫者。

這樣，如果不同的處理程序最終進行類似的計算並具有類似的狀態，則這些狀態的大部分將被共享，即使它們沒有如 **3.2.4** 中所述透過函數 f 的應用直接關聯。

3.2.6. 最佳化技術的重要性. 在同一 catchain 內用於共享不同內部狀態部分的最佳化技術 **3.2.4** 和 **3.2.5** 對於改善大型處理程序群組中 BCM 的記憶體配置和性能至關重要。在 $N \approx 100$ 個處理程序的群組中，改進幅度達到數個數量級。沒有這些最佳化，BCM 將不適合其預期目的（對 TON 區塊鏈中驗證者生成的新區塊進行 BFT 共識）。

3.2.7. 訊息 m 包含狀態 $\sigma_{D_m^+}$ 的雜湊. 每個訊息 m 都包含對應狀態 $\sigma_{D_m^+}$（的抽象表示）的（Merkle）雜湊。非常粗略地說，此雜湊是使用 **3.2.1** 的節點樹表示遞迴計算的：節點內的所有節點引用都被替換為被引用節點的（遞迴計算的）雜湊，並計算結果位元組序列的簡單 64 位元雜湊。此雜湊也用於如 **3.2.5** 中所述的記憶化。

訊息中此欄位的目的是為不同處理程序（以及可能由狀態更新演算法的不同實作）執行的 $\sigma_{D_m^+}$ 計算提供健全性檢查：一旦為新傳遞的訊息 m 計算出 $\sigma_{D_m^+}$，計算出的 $\sigma_{D_m^+}$ 的雜湊就會與儲存在 m 標頭中的值進行比較。如果這些值不相等，則會將錯誤訊息輸出到錯誤日誌中（軟體不會採取進一步行動）。可以檢查這些錯誤日誌以偵測 BCP 不同版本之間的錯誤或不相容性。

3.3. 重新啟動或崩潰後的狀態恢復. BCP 通常會使用 catchain 幾分鐘；在此期間，運行 Catchain 協議的程式（驗證者軟體）可能會被終止並重新啟動，可能是故意的（例如，由於計劃的軟體更新），也可能是無意的（程式可能由於此子系統或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最後一次重新啟動後未建立的所有 catchain。然而，這將導致一些驗證者在幾分鐘內（直到建立下一個 catchain 實例）不參與建立任何區塊，這是不希望出現的。因此，每次重新啟動後都會運行 catchain 狀態恢復協議，以便驗證者可以繼續參與同一 catchain。

3.3.1. 所有已傳遞訊息的資料庫. 為此，為每個活躍的 catchain 建立一個特殊的資料庫。此資料庫包含所有已知和已傳遞的訊息，按其識別符（雜

湊）索引。簡單的鍵值資料庫就足以滿足此目的。當前處理程序 i 生成的最新出站訊息 $m = m_{i,s}$ 的雜湊也儲存在此資料庫中。重新啟動後，所有直到 m 的訊息都會按正確順序遞迴傳遞（就像所有這些訊息剛從網路以任意順序接收一樣）並由較高層級協議處理，直到最終傳遞 m，從而恢復當前狀態。

3.3.2. 將新訊息刷新到磁碟. 我們已經在 **2.6** 中解釋過，新建立的訊息儲存在所有已傳遞訊息的資料庫中（參見 **3.3.1**），並且在將新訊息發送給所有網路鄰居節點之前，資料庫會刷新到磁碟。這樣，我們可以確保如果系統崩潰並重新啟動，訊息不會遺失，從而避免建立非自願的分叉。

3.3.3. 避免重新計算狀態 $\sigma_{D_m^+}$. 實作可能使用如 **3.2.3** 中所述的包含所有先前計算狀態的僅附加檔案，以避免在重新啟動後重新計算所有狀態，用磁碟空間換取計算能力。然而，當前實作不使用此最佳化。

3.4. 區塊共識協議的高階描述. 現在我們準備呈現 TON 區塊鏈驗證者用於生成新區塊鏈區塊並在其上達成共識的區塊共識協議的高階描述。本質上，它是一個在 catchain（Catchain 協議的實例）之上運行的三階段提交協議，catchain 被用作處理程序群組中的「強化」訊息廣播系統。

3.4.1. 建立新的 catchain 訊息. 回想一下，較低層級的 Catchain 協議不會自行建立廣播訊息（唯一的例外是包含分叉證明的服務廣播，參見 **2.7.5**）。相反，當需要建立新訊息時，會透過呼叫回呼來要求較高層級協議（BCP）執行此操作。除此之外，建立新訊息還可能由當前虛擬狀態的變化和計時器警報觸發。

3.4.2. Catchain 訊息的有效負載. 這樣，catchain 訊息的有效負載總是由較高層級協議（例如 BCP）決定。對於 BCP，此有效負載包含

- 當前 Unix 時間。它在同一處理程序的後續訊息上必須是非遞減的。（如果違反此限制，所有處理此訊息的處理程序將默默地用同一發送者的前一個訊息中看到的最大 Unix 時間替換此 Unix 時間。）
- 若干（零個或多個）下列允許型別之一的 *BCP* 事件。

3.4.3. BCP 事件. 我們剛剛解釋過 catchain 訊息的有效負載包含若干（可能為零）BCP 事件。現在我們列出所有允許的 BCP 事件型別。

- Submit(*round*, *candidate*) — 建議新的區塊候選
- Approve(*round*, *candidate*, *signature*) — 區塊候選已通過本地驗證

- Reject(*round*, *candidate*) — 區塊候選未通過本地驗證
- CommitSign(*round*, *candidate*, *signature*) — 區塊候選已被接受並簽章
- Vote(*round*, *candidate*) — 對區塊候選的投票
- VoteFor(*round*, *candidate*) — 必須在此輪次中投票給此區塊候選（即使當前處理程序有不同意見）
- PreCommit(*round*, *candidate*) — 對區塊候選的初步承諾（用於三階段提交方案）

3.4.4. 協議參數. BCP 的若干參數必須預先固定（在 catchain 的創世訊息中，它們從當前主鏈狀態中提取的配置參數值初始化）：

- K — 一次嘗試的持續時間（以秒為單位）。在當前實作中它是整數秒數；然而，這是實作細節，不是協議的限制
- Y — 接受候選的快速嘗試次數
- C — 一輪中建議的區塊候選數
- Δ_i（對於 $1 \leq i \leq C$）— 建議優先級為 i 的區塊候選之前的延遲
- Δ_∞ — 批准空候選之前的延遲

這些參數的可能值為 $K = 8$、$Y = 3$、$C = 2$、$\Delta_i = 2(i-1)$、$\Delta_\infty = 2C$。

3.4.5. 協議概述. BCP 由在同一 catchain 內執行的若干輪次組成。在某個時間點可能有多個輪次處於活躍狀態，因為一個輪次的某些階段可能與其他輪次的其他階段重疊。因此，所有 BCP 事件都包含明確的輪次識別符 *round*（從零開始的小整數）。每個輪次要麼透過（集體）接受由參與處理程序之一建議的區塊候選而終止，要麼透過接受特殊的空候選而終止——這是一個虛擬值，表示沒有接受真正的區塊候選，例如因為根本沒有建議區塊候選。在輪次終止後（從參與處理程序的角度來看），即一旦區塊候選收集到超過 2/3 所有驗證者的 CommitSign 簽章，就只能向該輪次添加 CommitSign 事件；處理程序自動開始參與下一輪（使用下一個識別符）並忽略所有具有不同 *round* 值的 BCP 事件。[6]

[6]這也意味著每個處理程序隱含地決定下一輪開始的 Unixtime，並從這個時間開始計算所有延遲，例如區塊候選提交延遲。

每個輪次細分為若干嘗試。每次嘗試持續預定的 K 秒時間段（BCP 使用時鐘來測量時間和時間間隔，並假設「好」處理程序的時鐘或多或少彼此一致；因此，BCP 不是非同步 BFT 協議）。每次嘗試在 Unixtime 恰好可被 K 整除時開始，並持續 K 秒。嘗試識別符 *attempt* 是其開始時的 Unixtime 除以 K。因此，嘗試或多或少由 32 位元整數連續編號，但不是從零開始。一輪的前 Y 次嘗試是快速的；其餘嘗試是慢速的。

3.4.6. 嘗試識別。快速和慢速嘗試. 與輪次相反，BCP 事件沒有參數來指示它們所屬的嘗試。相反，此嘗試是由包含 BCP 事件的 catchain 訊息有效負載中指示的 Unix 時間隱含決定的（參見 **3.4.2**）。此外，嘗試細分為快速（處理程序參與的輪次的前 Y 次嘗試）和慢速（同一輪次的後續嘗試）。此細分也是隱含的：處理程序在輪次中發送的第一個 BCP 事件屬於某個嘗試，從這個嘗試開始的 Y 次嘗試被此處理程序視為快速的。

3.4.7. 區塊生產者和區塊候選. 每輪中有 C 個指定的區塊生產者（成員處理程序）。這些區塊生產者的（有序）列表由確定性演算法計算（在最簡單的情況下，第 i 輪使用處理程序 i、$i+1$、…、$i+C-1$，索引取模 N，即 catchain 中處理程序的總數），所有參與者無需任何額外的通訊或協商即可知道。處理程序在此列表中按優先級遞減排序，因此列表的第一個成員具有最高優先級（即，如果它及時建議區塊候選，則此區塊候選很有可能被協議接受）。

第一個區塊生產者可以在輪次開始後立即建議區塊候選。其他區塊生產者只能在一些延遲 Δ_i 之後建議區塊候選，其中 i 是生產者在指定區塊生產者列表中的索引，且 $0 = \Delta_1 \leq \Delta_2 \leq \ldots$。在從輪次開始經過一些預定時間段 Δ_∞ 後，自動假設建議了特殊的空候選（即使沒有明確的 BCP 事件指示此點）。因此，一輪中最多建議 $C+1$ 個區塊候選（包括空候選）。

3.4.8. 建議區塊候選. TON 區塊鏈的區塊候選由兩個大型「檔案」組成——區塊和整理資料，以及一個小標頭，包含正在生成的區塊的描述（最重要的是，區塊候選的完整區塊識別符，包含工作鏈和分片識別符、區塊序號、其檔案雜湊和其根雜湊）和兩個大型檔案的 sha256 雜湊。此小標頭的僅一部分（包括兩個檔案的雜湊和其他重要資料）在 BCP 事件（例如 Submit 或 CommitSign）中用作 *candidate* 來引用特定的區塊候選。大量資料（最重要的是兩個大型檔案）透過為此目的在 ADNL 上實作的串流廣播協議在與 catchain 關聯的覆蓋網路中傳播（參見 [3, 5]）。此大量資料傳播機制對於共識協議的有效性並不重要（唯一重要的是大型檔案的雜湊是 BCP 事件的一部分，因此是 catchain 訊息的一部分，在那裡它們由發送者簽章，並且在任何參與節點接收到大型檔案後檢查這些雜湊；因此，沒有人可以替換或破壞這些檔案）。區塊生產者在 catchain 中建立

Submit(*round*, *candidate*) BCP 事件，與區塊候選的傳播並行，指示此區塊生產者提交此特定區塊候選。

3.4.9. 處理區塊候選. 一旦處理程序在已傳遞的 catchain 訊息中觀察到 Submit BCP 事件，它就會檢查此事件的有效性（例如，其發起處理程序必須在指定生產者列表中，且當前 Unixtime 必須至少為輪次開始時間加上最小延遲 Δ_i，其中 i 是此生產者在指定生產者列表中的索引），如果有效，則將其記住在當前 catchain 狀態中（參見 **3.1**）。之後，當接收到包含與此區塊候選關聯的檔案（具有正確雜湊值）的串流廣播時（或者，如果這些檔案已經存在，則立即），處理程序呼叫驗證者實例來驗證新的區塊候選（即使此區塊候選是由此處理程序本身建議的 !）。根據此驗證的結果，建立 Approve(*round*, *candidate*, *signature*) 或 Reject(*round*, *candidate*) BCP 事件（並嵌入新的 catchain 訊息中）。請注意，Approve 事件中使用的 *signature* 使用最終將用於簽章已接受區塊的相同私鑰，但簽章本身與 CommitSign 中使用的簽章不同（實際簽章的是具有不同魔術數字的結構的雜湊）。因此，此臨時簽章不能用於向外部觀察者偽造此特定驗證者處理程序對此區塊的接受。

3.4.10. 一輪的概述. BCP 的每一輪進行如下：

- 在輪次開始時，若干處理程序（來自預定的指定生產者列表）提交其區塊候選（根據其生產者優先級有一定延遲）並透過 Submit 事件（併入 catchain 訊息中）反映此事實。
- 一旦處理程序接收到提交的區塊候選（即觀察到 Submit 事件並透過共識協議外部的手段接收所有必要的檔案），它就會開始驗證此候選，並最終為此區塊候選建立 Approve 或 Reject 事件。
- 在每次快速嘗試（即前 Y 次嘗試之一）期間，每個處理程序要麼投票給已收集到超過 2/3 所有處理程序投票的區塊候選，要麼，如果尚無此類候選，則投票給具有最高優先級的有效（即被超過 2/3 所有處理程序 Approve 的）區塊候選。投票透過建立 Vote 事件（嵌入新 catchain 訊息中）執行。
- 在每次慢速嘗試（即除前 Y 次之外的任何嘗試）期間，每個處理程序要麼投票給之前（由同一處理程序）PreCommit 的候選，要麼投票給由 VoteFor 建議的候選。
- 如果區塊候選在當前嘗試期間已從超過 2/3 所有處理程序接收到投票，且當前處理程序觀察到這些投票（它們在 catchain 狀態中收集），則建立 PreCommit 事件，表示處理程序將來只會投票給此候選。

- 如果區塊候選在一次嘗試內從超過 2/3 所有處理程序收集到 PreCommit，則假設它被（群組）接受，觀察到這些 PreCommit 的每個處理程序都會建立帶有有效區塊簽章的 CommitSign 事件。這些區塊簽章在 catchain 中註冊，最終被收集以建立「區塊證明」（包含超過 2/3 驗證者對此區塊的簽章）。此區塊證明是共識協議的外部輸出（連同區塊本身，但不包括其整理資料）；它最終在已訂閱此分片（或主鏈）新區塊的所有全節點的覆蓋網路中傳播。

- 一旦區塊候選從超過 2/3 所有驗證者收集到 CommitSign 簽章，則輪次被視為完成（至少從觀察到所有這些簽章的處理程序的角度來看）。之後，此處理程序只能向該輪次添加 CommitSign，且處理程序自動開始參與下一輪（並忽略與其他輪次相關的所有事件）。

請注意，上述協議可能導致驗證者簽章（在 CommitSign 事件中）之前由同一驗證者 Reject 的區塊候選（這是一種「服從多數意志」）。

3.4.11. Vote 和 PreCommit 訊息是確定性建立的. 請注意，每個處理程序在每次嘗試中最多可以建立一個 Vote 和最多一個 PreCommit 事件。此外，這些事件完全由包含此類事件的 catchain 訊息 m 的發送者的狀態 σ_{D_m} 決定。因此，接收者可以偵測無效的 Vote 或 PreCommit 事件並忽略它們（從而緩解其他參與者的拜占庭行為）。另一方面，可能會接收到根據對應狀態 σ_{D_m} 應該包含 Vote 或 PreCommit 事件但實際上不包含的訊息 m。在這種情況下，當前實作會自動建立缺失的事件，並繼續進行，就好像 m 從一開始就包含它們一樣。然而，此類拜占庭行為實例要麼被糾正，要麼被忽略（並將訊息輸出到錯誤日誌中），但違規處理程序不會受到其他懲罰（因為這需要非常大的不當行為證明，而無法存取 catchain 內部狀態的外部觀察者無法獲得這些證明）。

3.4.12. 同一處理程序的多個 Vote 和 PreCommit. 請注意，處理程序通常會忽略同一發起處理程序在同一嘗試內生成的後續 Vote 和 PreCommit，因此通常處理程序最多只能投票給一個區塊候選。然而，可能發生「好」處理程序間接觀察到由拜占庭處理程序建立的分叉，該分叉的不同分支中有對不同區塊候選的 Vote（如果「好」處理程序從兩個之前未看到此分叉的其他「好」處理程序那裡得知這兩個分支，就會發生這種情況）。在這種情況下，兩個 Vote（對不同候選）都被考慮在內（添加到當前處理程序的合併狀態中）。類似的邏輯適用於 PreCommit。

3.4.13. 批准或拒絕區塊候選. 請注意，區塊候選在被 Submit 之前不能被 Approve 或 Reject（即，未在之前有對應 Submit 事件的 Approve 事件將被忽略），並且候選在達到其提交的最短時間（輪次開始時間加上依

賴於優先級的延遲 Δ_i）之前不能被批准，即任何「好」處理程序都會將其 Approve 的建立推遲到此時間。此外，在同一輪中不能 Approve 同一生產者的多個候選（即，即使處理程序 Submit 了若干候選，其他「好」處理程序也只會 Approve 其中一個——據推測是第一個；像往常一樣，這意味著「好」處理程序在接收時會忽略後續的 Approve 事件）。

3.4.14. 批准空區塊候選. 一旦從輪次開始經過延遲 Δ_∞，所有（好的）處理程序也會明確批准隱含的空區塊候選（透過建立 Approve 事件）。

3.4.15. 選擇區塊候選進行投票. 每個處理程序透過應用以下規則（按呈現順序）選擇可用區塊候選之一（包括隱含的空候選）並為此候選投票（透過建立 Vote 事件）：

- 如果當前處理程序在之前的嘗試之一期間為候選建立了 PreCommit 事件，並且自那時以來沒有其他候選從超過 2/3 所有處理程序收集到投票（即在後續嘗試之一內，包括到目前為止的當前嘗試；在這種情況下我們說 PreCommit 事件仍然是活躍的），則當前處理程序再次投票給此候選。

- 如果當前嘗試是快速的（即從當前處理程序的角度來看是輪次的前 Y 次嘗試之一），並且候選在當前或之前嘗試之一期間從超過 2/3 所有處理程序收集到投票，則當前處理程序投票給此候選。在平局的情況下，選擇來自所有此類嘗試中最新嘗試的候選。

- 如果當前嘗試是快速的，並且前面的規則不適用，則處理程序投票給所有合格候選中具有最高優先級的候選，即從超過 2/3 所有處理程序收集到 Approve（當前處理程序可觀察到）的候選。

- 如果當前嘗試是慢速的，則處理程序僅在收到同一嘗試中的有效 VoteFor 事件後才投票。如果第一條規則適用，則處理程序根據它投票（即投票給之前 PreCommit 的候選）。否則它投票給 VoteFor 事件中提及的區塊候選。如果有若干此類有效事件（在當前嘗試期間），則選擇具有最小雜湊的候選（這可能發生在與在分叉的不同分支中建立的不同 VoteFor 事件相關的罕見情況中，參見 **3.4.12**）。

「空候選」被認為具有最低優先級。它也需要明確的 Approve 才能被投票（前兩條規則除外）。

3.4.16. 在慢速嘗試期間建立 VoteFor 事件. VoteFor 事件由協調者在慢速嘗試開始時建立——在參與 catchain 的所有處理程序的有序列表中索引為 $attempt \bmod N$ 的處理程序（像往常一樣，這意味著由另一個處理程

序建立的 VOTEFOR 將被所有「好」處理程序忽略）。此 VOTEFOR 事件引用已從超過 2/3 所有處理程序收集到 APPROVE 的區塊候選（包括空候選）之一，通常從所有此類候選中隨機選擇。本質上，這是向所有沒有活躍 PRECOMMIT 的其他處理程序建議投票給此區塊候選。

3.5. BCP 的有效性. 現在我們呈現在 **3.4** 中描述的 TON 區塊共識協議（BCP）有效性證明的概要，假設少於三分之一的所有處理程序表現出拜占庭（任意惡意的，可能違反協議的）行為，這是拜占庭容錯協議的慣例。在此小節中，我們僅考慮細分為若干嘗試的 BCP 的一輪。

3.5.1. 基本假設. 讓我們再次強調，我們假設少於三分之一的所有處理程序是拜占庭的。所有其他處理程序被假設為好的，即它們遵循協議。

3.5.2. 加權 BCP. 本小節中的推理對於加權變體的 *BCP* 也是有效的。在此變體中，每個處理程序 $i \in I$ 被預先分配正權重 $w_i > 0$（在 catchain 的創世訊息中固定），關於「超過 2/3 所有處理程序」和「少於三分之一所有處理程序」的陳述被理解為「按權重計超過 2/3 所有處理程序」，即「具有總權重 $\sum_{j\in J} w_j > \frac{2}{3}\sum_{i\in I} w_i$ 的處理程序子集 $J \subset I$」，第二個屬性類似。特別是，我們的「基本假設」**3.5.1** 應理解為「所有拜占庭處理程序的總權重小於所有處理程序總權重的三分之一」。

3.5.3. 有用的不變量. 我們在此收集 BCP 的一輪期間（在 catchain 內部）所有 BCP 事件遵守的一些有用不變量。這些不變量以兩種方式強制執行。首先，任何「好」（非拜占庭）處理程序不會建立違反這些不變量的事件。其次，即使「壞」處理程序建立違反這些不變量的事件，所有「好」處理程序在包含此事件的 catchain 訊息傳遞給 BCP 時將偵測到此情況並忽略此類事件。即使在這些預防措施之後，仍存在與分叉相關的一些可能問題（參見 **3.4.12**）；我們分別指出如何解決這些問題，並在此列表中忽略它們。因此：

- 每個處理程序（在 BCP 的一輪內）最多有一個 SUBMIT 事件。
- 每個處理程序針對一個候選最多有一個 APPROVE 或 REJECT 事件（更準確地說，即使同一指定區塊生產者建立多個候選，其中只有一個可以被另一個處理程序 APPROVE）。[7]這是透過要求所有「好」處

[7]事實上，REJECT 僅出現在此限制中，並不影響任何其他內容。因此，任何處理程序可以放棄發送 REJECT 而不違反協議，並且可以完全從協議中移除 REJECT 事件。相反，協議的當前實作仍然生成 REJECT，但不會在接收時檢查任何內容，也不會在 catchain 狀態中記住它們。只會輸出訊息到錯誤日誌，並且有問題的候選將儲存到特殊目錄以供將來研究，因為 REJECT 通常表示拜占庭對手的存在，或者建議區塊的節點或建立 REJECT 事件的節點上的整理者（區塊生成）或驗證者（區塊驗證）軟體中存在錯誤。

理程序忽略（即不為其建立 Approve 或 Reject）由同一生產者建議但不是他們了解到的第一個候選的所有候選來實現的。

- 每個處理程序在每次嘗試期間最多有一個 Vote 和最多一個 PreCommit 事件。
- 每次（慢速）嘗試期間最多有一個 VoteFor 事件。
- 每個處理程序最多有一個 CommitSign 事件。
- 在慢速嘗試期間，每個處理程序要麼為其先前 PreCommitted 的候選投票，要麼為此嘗試的 VoteFor 事件中指示的候選投票。

可以透過在適當位置添加「有效」一詞來稍微改進上述陳述（例如，最多有一個有效的 Submit 事件...）。

3.5.4. 更多不變量.

- 每個指定生產者最多有一個合格候選（即從超過 2/3 所有處理程序收到 Approve 的候選），其他生產者沒有合格候選。
- 總共最多有 $C + 1$ 個合格候選（來自 C 個指定生產者的最多 C 個候選，加上空候選）。
- 候選只有在同一嘗試期間收集到超過 2/3 的 PreCommit 時才能被接受（更準確地說，候選只有在超過 2/3 所有處理程序為此候選建立 PreCommit 事件且屬於同一嘗試時才被接受）。
- 候選只有在它是合格候選時才能被 Vote、PreCommitted 或在 VoteFor 中提及，這意味著它先前已從超過 2/3 所有驗證者收集到 Approve（即只有在先前由超過 2/3 所有處理程序建立此候選的 Approve 事件並在包含 Vote 事件的訊息可觀察到的 catchain 訊息中註冊時，才能為候選建立有效的 Vote 事件，PreCommit 和 VoteFor 事件類似）。

3.5.5. 最多接受一個區塊候選. 現在我們聲稱最多可以接受一個區塊候選（在 *BCP* 的一輪中）。實際上，候選只有在同一嘗試內從超過 2/3 所有處理程序收集到 PreCommit 時才能被接受。因此，兩個不同的候選不能在同一嘗試期間實現此目標（否則超過三分之一的所有驗證者必須在一次嘗試內為兩個不同的候選建立 PreCommit，從而違反上述不變量；但我們已假設少於三分之一的所有驗證者表現出拜占庭行為）。現在假設兩個不

同的候選 c_1 和 c_2 在兩個不同的嘗試 a_1 和 a_2 中從超過 2/3 所有處理程序收集到 PreCommit。我們可以假設 $a_1 < a_2$。根據 **3.4.15** 的第一條規則，在嘗試 a_1 期間為 c_1 建立 PreCommit 的每個處理程序必須在所有後續嘗試 $a' > a_1$ 中繼續為 c_1 投票，或至少不能為任何其他候選投票，除非另一個候選 c' 在後續嘗試期間從超過 2/3 所有處理程序收集到 Vote（並且即使某些處理程序試圖不為 c_1 建立這些新的 Vote 事件，這個不變量也會強制執行，參見 **3.4.11**）。因此，如果 $c_2 \neq c_1$ 在嘗試 $a_2 > a_1$ 期間收集到必要數量的 PreCommit，則至少有一次嘗試 a'，$a_1 < a' \leq a_2$，使得某個 $c' \neq c_1$（不一定等於 c_2）在嘗試 a' 期間從超過 2/3 所有處理程序收集到 Vote。讓我們固定最小的這樣的 a'，以及在嘗試 a' 期間收集到許多投票的相應 $c' \neq c_1$。超過 2/3 所有驗證者在嘗試 a' 期間為 c' 投票，超過 2/3 所有驗證者在嘗試 a_1 期間為 c_1 進行 PreCommitted，並且由於 a' 的最小性，不存在嘗試 a'' 使得 $a_1 < a'' < a'$，使得與 c_1 不同的候選在嘗試 a'' 期間從超過 2/3 所有投票中收集到。因此，所有為 c_1 進行 PreCommitted 的驗證者在嘗試 a' 期間只能為 c_1 投票，同時我們假設 c' 在同一嘗試 a' 期間從超過 2/3 所有驗證者收集到投票。這意味著超過 1/3 所有驗證者在此嘗試期間以某種方式同時為 c_1 和 c' 投票（或為 c' 投票而他們只能為 c_1 投票），即超過 1/3 所有驗證者表現出拜占庭行為。根據我們的基本假設 **3.5.1**，這是不可能的。

3.5.6. 一次嘗試期間最多一個區塊候選可被 PreCommitted. 請注意，透過與 **3.5.5** 第一部分中相同的推理，在同一嘗試內建立的所有有效 PreCommit 事件（如果有的話）必須引用同一個區塊候選：由於候選 c 的有效 PreCommit 事件只能在觀察到同一嘗試內超過 2/3 所有處理程序為此候選投票後建立（並且所有好的處理程序會忽略無效的 PreCommit），同一嘗試內不同候選 c_1 和 c_2 的有效 PreCommit 事件的存在將意味著超過三分之一的所有處理程序在此嘗試內同時為 c_1 和 c_2 投票，即它們表現出拜占庭行為。鑑於我們的基本假設 **3.5.1**，這是不可能的。

3.5.7. 透過觀察到更新的 PreCommit 來停用先前的 PreCommit. 我們聲稱每當具有活躍 *PreCommit* 的處理程序觀察到任何處理程序在稍後嘗試中為不同候選建立的有效 *PreCommit* 時，其先前活躍的 *PreCommit* 將被停用。回想一下，我們說處理程序具有活躍 *PreCommit*，如果它在某個嘗試 a 期間為某個候選 c 建立了 PreCommit，在任何嘗試 $a' > a$ 期間沒有建立任何 PreCommit，並且在任何嘗試 $a' > a$ 期間沒有觀察到超過 2/3 所有驗證者為任何候選 $\neq c$ 投票。任何處理程序最多有一個活躍 PreCommit，如果它有一個，它必須只為預提交的候選投票。

現在我們看到，如果自嘗試 a 以來對候選 c 有活躍 PreCommit 的處理程序觀察到在某個稍後嘗試 $a' > a$ 期間為候選 c' 建立的有效 PreCommit

(通常由另一個處理程序建立)，那麼第一個處理程序還必須觀察到包含較新 PreCommit 的訊息的所有依賴項；這些依賴項必然包括在同一嘗試 $a' > a$ 期間為同一候選 $c' \neq c$ 建立的超過 2/3 所有驗證者的有效 Vote (因為否則較新的 PreCommit 將無效，並將被其他處理程序忽略)；根據定義，觀察到所有這些 Vote 會停用原始 PreCommit。

3.5.8. 證明協議收斂性的假設. 現在我們將在一些假設下證明上述協議收斂 (即在接受區塊候選後終止) 的概率為一，這些假設本質上告訴我們有足夠多的「好」處理程序 (即勤奮遵循協議且不會在發送新訊息之前引入任意延遲的處理程序)，並且這些好的處理程序至少不時享有良好的網路連接。更準確地說，我們的假設如下：

- 存在由「好」處理程序組成的子集 $I^+ \subset I$，包含超過 2/3 所有處理程序。
- 來自 I^+ 的所有處理程序具有良好同步的時鐘 (相差最多 τ，其中 τ 是下面描述的網路延遲界限)。
- 如果有無限多次嘗試，則無限多次嘗試對於來自 I^+ 的處理程序之間的網路連接是「好的」，這意味著在此嘗試或更早期間由來自 I^+ 的處理程序建立的所有訊息在建立後最多 $\tau > 0$ 秒內以至少 $q > 0$ 的概率傳遞給來自 I^+ 的任何其他處理程序，其中 $\tau > 0$ 和 $0 < q < 1$ 是一些固定參數，使得 $5\tau < K$，其中 K 是一次嘗試的持續時間。
- 此外，如果協議運行無限多次嘗試，則任何等差數列嘗試都包含上述意義上的無限多次「好」嘗試。
- 來自 I^+ 的處理程序在慢速嘗試期間在慢速嘗試開始後某個固定或隨機延遲後建立 VoteFor，使得此延遲屬於區間 $(\tau, K - 3\tau)$，概率至少為 q'，其中 $q' > 0$ 是固定參數。
- 來自 I^+ 的處理程序，當輪到它成為慢速嘗試的協調者時，在所有合格候選 (即從超過 2/3 所有驗證者收集到 Approve 的候選) 中均勻隨機選擇 VoteFor 的候選。

3.5.9. 在這些假設下協議終止. 現在我們聲稱在 **3.5.8** 中列出的假設下，上述 *BCP* 協議 (的每一輪) 以概率一終止。證明如下進行。

- 讓我們假設協議不收斂。那麼它將永遠繼續運行。我們將忽略前幾次嘗試，只考慮從某個 a_0 開始的嘗試 a_0、$a_0 + 1$、$a_0 + 2$、...，稍後選擇。

- 由於來自 I^+ 的所有處理程序繼續參與協議，它們將在輪次開始後不久至少建立一條訊息（每個處理程序對此的感知可能略有不同）。例如，它們將在輪次開始後不超過 Δ_∞ 秒為空候選建立 Approve。因此，它們將在之後最多 KY 秒考慮所有嘗試為慢速。透過適當選擇 a_0，我們可以假設我們考慮的所有嘗試從 I^+ 中所有處理程序的角度來看都是慢速的。

- 在「好」嘗試 $a \geq a_0$ 之後，來自 I^+ 的所有處理程序將看到來自 I^+ 所有其他處理程序為空候選建立的 Approve，並將從此認為空候選合格。由於有無限多次「好」嘗試，這將以概率一遲早發生。因此，我們可以假設（如有必要增加 a_0）從來自 I^+ 的所有處理程序的角度來看至少有一個合格候選，即空候選。

- 此外，將有無限多次嘗試 $a \geq a_0$ 被來自 I^+ 的所有處理程序視為慢速，具有來自 I^+ 的協調者，並且在 **3.5.8** 中定義的意義上「好」（關於網路連接）。讓我們稱這樣的嘗試為「非常好」。

- 考慮一次「非常好」的慢速嘗試 a。其協調者（屬於 I^+）將以 $q' > 0$ 的概率在建立其 VoteFor 事件之前等待 $\tau' \in (\tau, K - 3\tau)$ 秒。考慮來自 I^+ 的任何處理程序建立的最近 PreCommit 事件；讓我們假設它在嘗試 $a' < a$ 期間為某個候選 c' 建立。以 $qq' > 0$ 的概率，攜帶此 PreCommit 的 catchain 訊息將在其 VoteFor 事件生成時已經傳遞給協調者。在這種情況下，攜帶此 VoteFor 的 catchain 訊息將依賴於此 PreCommit(c') 事件，並且觀察到此 VoteFor 的所有「好」處理程序也將觀察到其依賴項，包括此 PreCommit(c')。我們看到以至少 qq' 的概率，在「非常好」的慢速嘗試期間接收到 *VoteFor* 事件的來自 I^+ 的所有處理程序也接收到最近的 *PreCommit*（如果有的話）。

- 接下來，考慮來自 I^+ 的任何接收到此 VoteFor 的處理程序，對於隨機選擇的合格候選 c，並假設已經存在一些 PreCommit，並且前述陳述成立。由於最多有 $C + 1$ 個合格候選（參見 **3.5.4**），以至少 $1/(C+1) > 0$ 的概率我們將有 $c = c'$，其中 c' 是最近 PreCommitted 的候選（根據 **3.5.6** 最多有一個這樣的候選）。在這種情況下，來自 I^+ 的所有處理程序將在此嘗試期間在收到此 VoteFor 後立即為 $c = c'$ 投票（以 qq' 的概率將在嘗試開始後不到 $K - 2\tau$ 秒內傳遞給任何處理程序 $j \in I^+$）。實際上，如果來自 I^+ 的處理程序 j 沒有活躍 PreCommit，它將為 VoteFor 中指示的值投票，即 c。如果 j 有活躍 PreCommit，並且它盡可能最近，即也在嘗試 a' 期間建

立，那麼它必須是對同一值 $c' = c$ 的 PreCommit（因為我們知道至少有一個嘗試 a' 期間對 c' 的有效 PreCommit，並且根據 **3.5.6** 嘗試 a' 期間所有其他有效 PreCommit 必須對同一 c'）。最後，如果 j 有來自嘗試 $< a'$ 的活躍 PreCommit，那麼一旦 VoteFor 及其所有依賴項（包括較新的 PreCommit(c')）傳遞給此處理程序 j（參見 **3.5.7**），它將變為非活躍，並且處理程序將再次為 VoteFor 中指示的值 c 投票。因此，來自 I^+ 的所有處理程序將在此嘗試期間為同一 $c = c'$ 投票，在嘗試開始後不到 $K - 2\tau$ 秒（以某個遠離零的概率）。

- 如果還沒有 PreCommit，則上述推理進一步簡化：接收到此 VoteFor 的來自 I^+ 的所有處理程序將立即為此 VoteFor 建議的候選 c 投票。

- 在兩種情況下，來自 I^+ 的所有處理程序將在嘗試開始後不到 $K - 2\tau$ 秒為同一候選 c 建立 Vote，並且這將以遠離零的正概率發生。

- 最後，來自 I^+ 的所有處理程序將接收到來自 I^+ 所有處理程序的這些對 c 的 Vote，再次在此嘗試開始後不到 $(K - 2\tau) + \tau = K - \tau$ 秒，即仍在同一嘗試期間（即使考慮到來自 I^+ 的處理程序之間的時鐘同步不完美）。這意味著它們都將為 c 建立有效的 PreCommit，即協議將在此嘗試期間以遠離零的概率接受 c。

- 由於有無限多次「非常好」的嘗試，並且每次這樣的嘗試期間成功終止的概率 $\geq p > 0$ 對於某個固定值 p，協議將以概率一成功終止。

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，可於 `http://pmg.csail.mit.edu/papers/osdi99.pdf` 取得。

[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。