Catchain Consensus: An Outline

Nikolai Durov

March 15, 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

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/[1]），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。

[1] 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)。[2] 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。

[2]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。[3]

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

[3]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. Note that the genesis block is not distributed among the participating nodes; it is rather computed independently by each participating node as explained in **2.2**. Since the hash of the genesis block is used as the catchain identifier (i.e., identifier of the specific instance of the Catchain protocol; cf. **2.3**), if a node (accidentally or intentionally) computes a different genesis block, it will be effectively locked out from participating in the "correct" instance of the protocol.

2.3.2. List of nodes participating in a catchain. Note that the (ordered) list of nodes participating in a catchain is fixed in the genesis block and hence it is known to all the participants and it is unambiguously determined by the hash of the genesis block (i.e., the catchain identifier), provided there are no (known) collisions for sha256. Therefore, we fix the number of participating nodes N in the discussion of one specific catchain below, and assume that the nodes are numbered from 1 to N (their real identities may be looked up in the list of participants using this index in range $1 \dots N$). The set of all participants will be denoted by I; we assume that $I = \{1 \dots N\}$.

2.4. Messages in a catchain. Catchain as a process group. One perspective is that a catchain is a *(distributed) process group* consisting of N known and fixed *(communicating) processes* (or *nodes* in the preceding terminology), and these processes generate *broadcast messages*, that are eventually broadcast to all members of the process group. The set of all processes is denoted by I; we usually assume that $I = \{1 \dots N\}$. The broadcasts generated by each process are numbered starting from one, so the n-th broadcast of process i will receive *sequence number* or *height* n; each broadcast should be uniquely determined by the identity or the index i of the originating process and its height n, so we can think of the pair (i, n) as the natural identifier of a broadcast message inside a process group.[4] The broadcasts generated by the same process i are expected to be delivered to every other process in exactly the same order they have been created, i.e., in increasing order of their height. In this respect a catchain is very similar to a process group in

[4]In the Byzantine environment of a catchain this is not necessarily true in all situations.

the sense of [1] or [7]. The principal difference is that a catchain is a "hardened" version of a process group tolerant to possible Byzantine (arbitrarily malicious) behavior of some participants.

2.4.1. Dependence relation on messages. One can introduce a *dependence relation* on all messages broadcast in a process group. This relation must be a strict partial order $\prec$, with the property that $m_{i,k} \prec m_{i,k+1}$, where $m_{i,k}$ denotes the k-th message broadcast by group member process with index i. The meaning of $m \prec m'$ is that m' *depends on* m, so that the (broadcast) message m' can be processed (by a member of the process group) only if m has been processed before. For instance, if the message m' represents the reaction of a group member to another message m, then it is natural to set $m \prec m'$. If a member of the process group receives a message m' before all its dependencies, i.e., messages $m \prec m'$, have been processed (or *delivered* to the higher-level protocol), then its processing (or *delivery*) is delayed until all its dependencies are delivered.

We have defined the dependence relation to be a strict partial order, so it must be transitive ($m'' \prec m'$ and $m' \prec m$ imply $m'' \prec m$), antisymmetric (at most one of $m' \prec m$ and $m \prec m'$ can hold for any two messages m and m') and anti-reflexive ($m \prec m$ never holds). If we have a smaller set of "basic dependencies" $m' \to m$, we can construct its transitive closure $\to^+$ and put $\prec := \to^+$. The only other requirement is that every broadcast of a sender depends on all previous broadcasts of the same sender. It is not strictly necessary to assume this; however, this assumption is quite natural and considerably simplifies the design of a messaging system inside a process group, so the Catchain protocol makes this assumption.

2.4.2. Dependence set or cone of a message. Let m be a (broadcast) message inside a process group as above. We say that the set $D_m := \{m' : m' \prec m\}$ is the *dependence set* or *dependence cone* of message m. In other words, D_m is the *principal ideal* generated by m in the partially ordered finite set of all messages. It is precisely the set of all messages that must be delivered before m is delivered.

2.4.3. Extended dependence cone of a message. We also define D_m^+, the *extended dependence cone of* m, by $D_m^+ := D_m \cup \{m\}$.

2.4.4. Cones, or ideals with respect to $\prec$. More generally, we say that a subset D of messages is a *cone* if it is an ideal with respect to the

dependence relation $\prec$, i.e., if $m \in D$ and $m' \prec m$ imply $m' \in D$. Of course, the dependence cone D_m and the extended dependence cone D_m^+ of any message m are cones (because any principal ideal in a partially ordered set is an ideal).

2.4.5. Identification of cones with the aid of vector time. Recall that we have assumed that any message depends on all preceding messages of the same sender, i.e. $m_{i,s} \prec m_{i,s+1}$ for any $i \in I$ and any $s > 0$, such that $m_{i,s+1}$ exists. This implies that any cone D is completely characterized by N values $\mathrm{VT}(D)_i$ indexed by $i \in I$:

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

(if no message $m_{i,s}$ is in D, we set $\mathrm{VT}(D)_i := 0$). Indeed, it is clear that

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

We say that the vector $\mathrm{VT}(D) = (\mathrm{VT}(D)_i)_{i\in I} \in \mathbb{N}_0^I$ with non-negative components $\mathrm{VT}(D)_i$ 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. We introduce a partial order $\leq$ on the set of all possible vector times $\mathbb{N}_0^I$, which is the product of the usual orders on $\mathbb{N}_0$:

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

It is immediate that $D \subset D'$ iff $\mathrm{VT}(D) \leq \mathrm{VT}(D')$; therefore, VT is a strict order-preserving embedding of the set of all cones contained in the set of all messages into $\mathbb{N}_0^I$.

2.4.7. Vector timestamp Vt(m) of a message m. Given any message m, we define its *vector timestamp* $\mathrm{VT}(m)$ as $\mathrm{VT}(D_m)$. In other words, message m can be delivered only after the first $\mathrm{VT}(m)_j$ messages generated by process j are delivered, and this is true for all $j \in I$.

If i is the sender of message m, and s is the height of message m, so that $m = m_{i,s}$, then $\mathrm{VT}(m)_i = s - 1$. We can define the *adjusted vector timestamp* $\mathrm{VT}^+(m)$ of message m by setting $VT^+(m)_j = VT(m)_j$ for $j \neq i$, $VT^+(m)_i = \mathrm{VT}(m)_i + 1 = s$. Alternatively, $\mathrm{VT}^+(m) = \mathrm{VT}(D_m^+)$, where $D_m^+ := D_m \cup \{m\}$ is the *extended dependence cone of* m (cf. **2.4.3**).

Note that $m' \preceq m$ iff $D^+_{m'} \subset D^+_m$ iff $\mathrm{VT}^+(m') \leq \mathrm{VT}^+(m)$ in $\mathbb{N}_0^I$, where $m' \preceq m$ means "$m' \prec m$ or $m' = m$". Similarly, $m' \prec m$ iff $D^+_{m'} \subset D_m$ iff $\mathrm{VT}^+(m') \leq \mathrm{VT}(m)$. In other words, *the dependence relation $\prec$ on (some or all) messages is completely determined by the adjusted vector timestamps of these messages.*

2.4.8. Using vector timestamps to correctly deliver broadcast messages. Vector timestamps can be used (in non-byzantine settings) to correctly deliver messages broadcast in a process group.[5] Namely, suppose that every broadcast message $m = m_{i,s}$ contains the index of its sender i and the vector timestamp of this message $\mathrm{VT}(m)$. Then each receiver j knows whether the message can be delivered or not. For this, j keeps track of the cone C_j of all messages delivered so far, for example by maintaining a *current timestamp* $\mathrm{VT}(j)$ equal to $\mathrm{VT}(C_j)$. In other words, $\mathrm{VT}(j)_k$ is the count of messages of sender k processed by j so far. If $\mathrm{VT}(m) \leq \mathrm{VT}(j)$, then the message m is delivered immediately and $\mathrm{VT}(j)$ is updated to $\sup(\mathrm{VT}(j), \mathrm{VT}^+(m))$ afterwards; this is equivalent to increasing $\mathrm{VT}(j)_i$ by one, where i is the original sender of message m. If this condition is not met, then m may be put into a waiting queue until $\mathrm{VT}(j)$ becomes large enough. Instead of passively waiting for the required broadcasts, j can construct the list of message indices (i', s') that are implicitly mentioned in $\mathrm{VT}(m)$ of some received but not delivered message m, and request messages with these indices from the neighbors from which j learned about m and $\mathrm{VT}(m)$; an alternative strategy (actually employed by the current implementation of the Catchain protocol) is to request these messages from randomly chosen neighbors from time to time. The latter strategy is simpler because it does not require remembering the immediate sources of all received messages (which may become unavailable anyway).

2.5. Message structure in a catchain. Catchain as a multi-blockchain. The message structure in a catchain is a bit more complicated than described above because of the necessity to support a BFT protocol. In particular, vector timestamps are not sufficient in a Byzantine setting. They have to be complemented by descriptions based on maximal elements of a dependence cone (such descriptions are typically used in non-byzantine settings

[5] We assume that all broadcast messages in the process group are "causal broadcasts" or "cbcast" in the terminology of [1], because we only need cbcasts for the implementation of Catchain protocol and Catchain consensus.

only when the process group is very large, so that vector timestamp sizes become prohibitive).

2.5.1. Describing cones by means of their maximal elements. An alternative way (to using a vector timestamp) of describing a message cone D is by listing all its *maximal elements* $\mathrm{Max}(D)$, i.e. elements $m \in D$, such that $m \prec m'$ does not hold for any $m' \in D$. Of course, one needs a suitable way of referring to messages without including them completely in order for this representation to be practical.

2.5.2. Message identifiers inside a catchain. Catchain protocol uses *sha256* hashes of (suitably serialized) messages as their unique identifiers. If we assume that there are no collisions for SHA256 (computable in reasonable, e.g., polynomial time), then a message m is completely identified within the process group by its hash $\mathrm{SHA256}(m)$.

2.5.3. Message headers. The header of a message $m = m_{i,s}$ inside a catchain (i.e., an instance of the Catchain protocol) always contains the index i of its sender, the height s, the catchain identifier (i.e., the hash of the genesis message, cf. **2.3**) and the set of hashes of maximal elements of the dependence cone of m, i.e., the set $\{\mathrm{SHA256}(m') : m' \in \mathrm{Max}(D_m)\}$. In particular, the hash $\mathrm{SHA256}(m_{i,s-1})$ of the previous message of the same sender is always included since $m_{i,s-1} \in \mathrm{Max}(D_m)$ if $s > 1$; for performance reasons, there is a separate field in the message header containing $\mathrm{SHA256}(m_{i,s-1})$. If $s = 1$, then there is no previous message, so the hash of the genesis message (i.e., the catchain identifier, cf. **2.3**) is used instead.

The vector timestamp $\mathrm{VT}(m)$ is not included in the message header; however, the header implicitly determines $\mathrm{VT}(m)$ since

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

Note that the message header is a part of the message, and in particular the hash of a message (i.e., the message identifier) depends on all data listed in the header. Therefore, we assume that the message identifier implicitly determines all the dependencies of the corresponding message (if there are no known collisions for SHA256).

2.5.4. Message signatures. Apart from that, every message in a catchain is signed by its creator. Since the list of participating nodes (processes) in

a catchain is known in advance, and this list includes the public keys of all processes, these message signatures can be checked by a receiving process immediately after a message is received. If the signature is invalid, the message is discarded without any further processing.

2.5.5. Message encryption. All messages in a catchain are also encrypted before being transferred from a node to its neighbor in the private overlay network underlying the catchain. However, this encryption is performed by lower-level network protocols (such as ADNL) and is not relevant to the discussion here. We would like to mention that correct encryption is possible here only because the list of participating processes includes not only the public keys of all processes, but also their ADNL addresses (which effectively are public encryption keys for network transmission).

Notice that even if the encryption had been absent, this would not violate the BFT properties of the protocol, because faking a message from another sender would not be possible because of the signatures. However, this might lead to a leak of information to outside observers, which is often undesirable.

2.5.6. Alternative perspective: a catchain as a multi-blockchain. Note that all messages created by the same sender i in a catchain turn out to have a simple "blockchain structure", because the header of $m_{i,s+1}$ contains the hash $\text{SHA256}(m_{i,s})$ (among other hashes of messages from $\text{Max}(D_{m_{i,s+1}})$) of the previous message of sender i. In this way each process i generates a simple blockchain consisting of its messages, with each "block" of this blockchain corresponding to one message and referring to the previous block by its hash, and sometimes includes references to blocks (i.e., messages) of other processes by mentioning the hashes of these blocks in its blocks. Each block is signed by its creator. The resulting structure is very similar to that of an "asynchronous payment channel" considered in [3, 5], but with N participants instead of 2.

2.6. Message propagation in a catchain. Now we are ready to describe message propagation in a catchain. Namely:

- The (lower-level) overlay network protocol maintains a list of neighbors in the private overlay network underlying the catchain and provides ADNL channels to each of these neighbors. This private overlay network has the same list of members (processes, nodes) as the catchain, and the neighbors of each node form an (oriented) subgraph on the

set of all participating nodes. This (essentially random) subgraph is strongly connected with probability very close to one.

- Each process generates some new messages from time to time (as needed by the higher-level protocol). These messages are augmented by catchain message headers as outlined in **2.5.3**, signed, and propagated to all known neighbors using the ADNL channels established by the overlay protocol.

- In contrast with the usual simple overlay broadcast protocol, the messages received from neighbors are not immediately rebroadcast to all other neighbors that are not known yet to have a copy of them. Instead, the signature is checked first, and invalid messages are discarded. Then the message is either delivered (if all its dependent messages have already been delivered), or put into a waiting queue. In the latter case, all the required messages mentioned in its header (i.e., the set $\mathrm{Max}(D_m)$) are pulled from the neighbor that sent this message (apart from that, attempts to download these missing messages from random neighbors are performed from time to time). If necessary, this process is repeated recursively until some messages can be delivered. Once a message is ready for local delivery (i.e., all its dependencies are already present), it is also rebroadcast to all neighbors in the overlay network.

- Apart from the recursive "pull" mechanism described above, a faster vector timestamp-based mechanism is also used, so that messages can be queried from neighbors by their senders and heights (learned from the vector timestamps of received messages). Namely, each process sends a special query containing the current vector timestamp to a randomly chosen neighbor from time to time. This peer-to-peer query leads to its receiver sending back all or some messages unknown to the sender (judging by their vector timestamps).

- This faster vector timestamp-based mechanism can be disabled for messages originating from certain senders as soon as a "fork" is detected, i.e., a second message with the same sender i and height s, but with a different hash, is learned from a neighbor, for example, during the fast or slow "pull" process. Once a fork created by i is detected, the corresponding component VT_i of all subsequent vector timestamps is

set to a special value ∞ to indicate that comparing the values of these components does not make sense anymore.

- When a message is delivered (to the higher-level protocol), this message is added into the cone C of processed messages of the current process (and the current vector timestamp is updated accordingly), and all subsequent messages generated by the current process will be assumed to depend on all the messages delivered so far (even if this is not logically necessary from the perspective of the higher-level protocol).

- If the set $\mathrm{Max}(C)$ of the maximal elements of the cone of processed messages becomes too large (contains more elements than a certain amount fixed in advance by the genesis message of the catchain), then the Catchain protocol asks the higher-level protocol to generate a new message (empty if no useful payload is available). After this new message is generated (and immediately delivered to the current process), C is updated and $\mathrm{Max}(C)$ consists of only one element (the new message). In this way the size of $\mathrm{Max}(C)$ and therefore the size of the message header always remain bounded.

- Once a message m is delivered and the set C is modified to include this message, a timer is set, and after some small delay the higher-level protocol is asked to create a new message (empty if necessary), so that this new message m^* would refer to the new C, similarly to the procedure described in the previous item. This new message m^* is pushed to all neighbors; since its header contains $\mathrm{Max}(C)$ for the new C, and $m \in C$, the neighbors learn not only about the newly-generated message m^*, but also about the original received message m. If some neighbors do not have a copy of m yet, they would require one (from the current process or not).

- All (broadcast) messages received and created in a catchain are stored into a special local database. This is especially important for newly-created messages (cf. **3.3.2**): if a message is created and sent to neighbors, but not saved into the database (and flushed to disk) before the creating process crashes and is restarted, then another message with the same sender and height can be created after restart, thus effectively leading to an involuntary "fork".

2.7. Forks and their prevention. One can see that the multi-blockchain structure of a catchain outlined above (with references to other blocks by their hashes and with signatures) leaves very little possibility for "cheating" in a consensus protocol built upon a catchain (i.e., using the catchain as a means for broadcasting messages inside a process group). The only possibility that is not detected immediately consists of creating two (or more) different versions of the same message $m_{i,s}$ (say, $m'_{i,s}$ and $m''_{i,s}$), and sending one version of this message $m'_{i,s}$ to some peers and a different version $m''_{i,s}$ to others. If s is minimal (for a fixed i), then this corresponds to a *fork* in blockchain terminology: two different next blocks $m'_{i,s}$ and $m''_{i,s}$ for the same previous block $m_{i,s-1}$.

Therefore, the Catchain protocol takes care to detect forks as soon as possible and prevent their propagation.

2.7.1. Detection of forks. The detection of forks is simple: if there are two different blocks $m'_{i,s}$ and $m''_{i,s}$ with the same creator $i \in I$ and the same height $s \geq 1$, and with valid signatures of i, then this is a fork.

2.7.2. Fork proofs. Block signatures in the Catchain protocol are created in such a way that creating *fork proofs* (i.e., the proof that a process i has intentionally created a fork) is especially simple since it is the hash of a very small structure (containing a magic number, the values of i and s, and the hash of the remainder of the message) that is actually signed. Therefore, only two such small structures and two signatures are required in a fork proof.

2.7.3. External punishment for creating forks. Notice that an external punishment for creating catchain forks may be used in the proof-of-stake blockchain generation context. Namely, the fork proofs may be submitted to a special smart contract (such as the elector smart contract of the TON Blockchain), checked automatically, and some part or all of the stake of the offending party may be confiscated.

2.7.4. Internal processing of forks. Once a fork (created by i) is detected (by another process j), i.e. j learns about two different messages $m_{i,s}$ and $m'_{i,s}$ created by i and having same height s (usually this happens while recursively downloading dependencies of some other messages), j starts ignoring i and all of its subsequent messages. They are not accepted and not broadcast further. However, messages created by i prior to the fork detection may be still downloaded if they are referred to in messages (blocks) created by

processes that did not see this fork before referring to such messages created by i.

2.7.5. Accepting messages from a "bad" process is bad. Furthermore, if process i learns about a fork created by process j, then i shows this to its neighbors by creating a new service broadcast message that contains the corresponding fork proof (cf. **2.7.2**). Afterwards, this and all subsequent messages of j cannot directly depend on any messages by the known "bad" producer i (but they still can refer to messages from another party k that directly or indirectly refer to messages of i if no fork by i was known to k at the time when the referring message was created). If j violates this restriction and creates messages with such invalid references, these messages will be discarded by all honest processes in the group.

2.7.6. The set of "bad" group members is a part of the intrinsic state. Each process i keeps its own copy of the set of known "bad" processes in the group, i.e., those processes that have created at least one fork or have violated **2.7.5**. This set is updated by adding j into it as soon as i learns about a fork created by j (or about a violation of **2.7.5** by j); after that, a callback provided by the higher-level protocol is invoked. This set is used when a new broadcast message arrives: if the sender is bad, then the message is ignored and discarded.

3 Block Consensus Protocol

We explain in this section the basic workings of the TON Block Consensus Protocol (cf. **1**), which builds upon the generic Catchain protocol (cf. **2**) to provide the BFT protocol employed for generating and validating new blocks of the TON Blockchain. The source code for the TON Block Consensus protocol resides in subdirectory `validator-session` of the source tree.

3.1. Internal state of the Block Consensus Protocol. The higher-level Block Consensus Protocol introduces a new notion to the catchain: that of an *internal state* of the Block Consensus Protocol (BCP), sometimes also (not quite correctly) called "the internal state of the catchain" or simply *catchain state*. Namely, each process $i \in I$ has a well-determined internal state σ_{C_i} after a subset of messages (actually always a dependence cone) C_i is delivered by the Catchain protocol to the higher-level protocol (i.e., to the Block Consensus Protocol in this case). Furthermore, this state $\sigma_{C_i} = \sigma(C_i)$ depends only on cone C_i, but not on the identity of the process $i \in I$, and can be defined for any dependence cone S (not necessarily a cone C_i of delivered messages for some process i at some point).

3.1.1. Abstract structure of the internal state. We start with an abstract structure of the internal state employed by BCP; more specific details will be provided later.

3.1.2. Updating the internal state. The Catchain protocol knows nothing about the internal state; it simply invokes appropriate callbacks supplied by the higher-level protocol (i.e., the BCP) whenever a message m is delivered. It is the job of the higher-level protocol to compute the new state $\sigma_{S'}$ starting from the previously computed state σ_S and the message m, where $S' = S \cup \{m\}$ (and necessarily $S \supset D_m$, otherwise m could not have been delivered at this point).

3.1.3. Recursive formula for updating the internal state. The abstract setup for computing σ_S for all cones S consists of three components:

- A value $\sigma_\emptyset$ for the initial state (this value actually depends on the genesis block of the catchain; we ignore this dependence here because we consider only one catchain at this point).

- A function f that computes the state $\sigma_{D_m^+}$ from the previous state σ_{D_m} and the newly-delivered message m:

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

  where D_m is the dependence cone of message m and $D_m^+ = D_m \cup \{m\}$ its extended dependence cone (cf. **2.4.3**). In most cases, f will actually satisfy the stronger condition

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

  However, this stronger condition is not required by the update algorithm.

- A "merge function" g that computes $\sigma_{S\cup T}$ from σ_S and σ_T:

$$\sigma_{S\cup T} = g(\sigma_S, \sigma_T) \quad \text{for any cones } S \text{ and } T \tag{7}$$

  (the union of two cones always is a cone). This function σ is applied by the update algorithm only in the specific case $T = D_m^+$ and $m \notin S$.

3.1.4. Commutativity and associativity of g. Note that (7) (for arbitrary cones S and T) implies associativity and commutativity of g, at least when g is applied to possible states (values of form σ_S for some cone S). In this respect g defines a commutative monoid structure on the set $\Sigma = \{\sigma_S : S \text{ is a cone}\}$. Usually g is defined or partially defined on a larger set $\tilde{\Sigma}$ of state-like values, and it may be commutative and associative on this larger set $\tilde{\Sigma}$, i.e., $g(x,y) = g(y,x)$ and $g(x, g(y,z)) = g(g(x,y), z)$ for x, y, $z \in \tilde{\Sigma}$ (whenever both sides of the equality are defined), with $\sigma_\emptyset$ as an unit, i.e., $g(x, \sigma_\emptyset) = x = g(\sigma_\emptyset, x)$ for $x \in \tilde{S}$ (under the same condition). However, this property, useful for the formal analysis of the consensus algorithm, is not strictly required by the state update algorithm, because this algorithm uses g in a deterministic fashion to compute σ_S.

3.1.5. Commutativity of f. Note that f, if it satisfies the stronger condition (6), must also exhibit a commutativity property

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

whenever S is a cone and m and m' are two messages with $D_m \subset S$, $D_{m'} \subset S$, $m \notin S$ and $m' \notin S$, because in this case $S\cup\{m\}$, $S\cup\{m'\}$ and $S\cup\{m, m'\}$

are also cones, and (6) implies that both sides of (8) are equal to $\sigma_{S\cup\{m,m'\}}$. Similarly to **3.1.4**, f is usually defined or partially defined on the product of a larger set $\tilde{\Sigma}$ of state-like values and of a set of message-like values; it may exhibit the "commutativity" property (8) or not on this larger set. If it does, this might be useful for formal analysis of the algorithms relying on σ_S, but this property is not strictly necessary.

3.1.6. The state update algorithm. The state update algorithm (independently executed by each process i) employed by the catchain (actually by the higher-level BCP) uses $\sigma_\emptyset$, f and g as follows:

- The algorithm keeps track of all $\sigma_{D_m^+}$ for all messages m delivered so far.
- The algorithm keeps track of σ_{C_i}, where C_i is the current dependence cone, i.e., the set of all messages m delivered (to the current process i). The initial value of σ_{C_i} is $\sigma_\emptyset$.
- When a new message m is delivered, the value of σ_{D^m} is computed by a repeated application of g since $D_m = \bigcup_{m'\in D_m} D^+_{m'} = \bigcup_{m'\in \mathrm{Max}(D_m)} D^+_{m'}$; therefore, if $\mathrm{Max}(D_m) = \{m'_1, \ldots, m'_k\}$, then

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

  The set $\mathrm{Max}(D_m)$ is explicitly listed in the header of message m in some fixed order $m'_1, \ldots, m'_k$; the above formula is applied with respect to this order (so the computation of D_m is deterministic). The first element in this list always is the previous message of the sender of m, i.e., if $m = m_{i,s+1}$, then $m'_1 = m_{i,s}$.
- After that, the value of $\sigma_{D_m^+}$ is computed by an application of f: $\sigma_{D_m^+} = f(\sigma_{D_m}, m)$. This value is memorized for future use.
- Finally, when a new message m is delivered to the current process i, thus updating C_i to $C'_i := C_i \cup \{m\}$, the algorithm uses the computed value $\sigma_{D_m^+}$ to update the current state

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

  This state, however, is "virtual" in the sense that it can be slightly changed later (especially if g is not commutative). Nevertheless, it is

used to make some important decisions by the higher-level algorithm (BCP).

- Once a new message m is generated and locally delivered, so that C_i becomes equal to D^+_m, the previously computed value of σ_{C_i} is discarded and replaced with $\sigma_{D^+_m}$ computed according to the general algorithm described above. If g is not commutative or not associative (for example, it may happen that $g(x, y)$ and $g(y, x)$ are different but equivalent representations of the same state), then this might lead to a slight change of the current "virtual" state of process i.

- If the lower-level (catchain) protocol reports to the higher-level protocol that a certain process $j \notin i$ is "bad" (i.e., j is found out to have created a fork, cf. **2.7.6**, or to have knowingly endorsed a fork by another process, cf. **2.7.5**), then the current (virtual) state σ_{C_i} is recomputed from scratch using the new set $C'_i = \bigcup_{m \in C_i,\ m \text{ was created by "good" process } k} D^+_m$ and the "merge" function g applied to the set of $\sigma_{D^+_m}$ where m runs through the set of last messages of the processes known to be good (or through the set of maximal elements of this set). The next created outbound message will depend only on the messages from C'_i.

3.1.7. Necessity to know the internal state of the other processes. Formula (9) implies that process i must also keep track of $\sigma_{D^+_m}$ for all messages m, created by this process or not. However, this is possible since these internal states are also computed by appropriate applications of the update algorithm. Therefore, BCP computes and remembers all $\sigma_{D^+_m}$ as well.

3.1.8. Function f would suffice. Notice that the update algorithm applies g only to compute $\sigma_{S \cup D^+_m} = g(\sigma_S, \sigma_{D^+_m})$ when S is a cone containing D_m, but not containing m. Therefore, every actual application of g could have been replaced by an application of f satisfying the extended property (6):

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

However, the update algorithm does not use this "optimization", because it would disable the more important optimizations described below in **3.2.4** and **3.2.5**.

3.2. The structure of the internal state. The structure of the internal state is optimized to make the *transition function* f of (5) and the *merge*

function g of (7) as efficiently computable as possible, preferably without the need of potentially unbounded recursion (just some loops). This motivates the inclusion of additional components into the internal state (even if these components are computable from the remainder of the internal state), which have to be stored and updated as well. This process of including additional components is similar to that employed while solving problems using dynamic programming, or to that used while proving statements by mathematical (or structural) induction.

3.2.1. The internal state is a representation of a value of an abstract algebraic data type. The internal representation of the internal state is essentially a (directed) tree (or rather a directed acyclic graph) or a collection of nodes; each node contains some immediate (usually integer) values and several pointers to other (previously constructed) nodes. If necessary, an extra *constructor tag* (a small integer) is added at the beginning of a node to distinguish between several possibilities. This structure is very similar to that used to represent values of abstract algebraic data types in functional programming languages such as Haskell.

3.2.2. The internal state is persistent. The internal state is *persistent*, in the sense that the memory used to allocate the nodes which are part of the internal state is never freed up while the catchain is active. Furthermore, the internal state of a catchain is actually allocated inside a huge contiguous memory buffer, and new nodes are always allocated at the end of the used portion of this buffer by advancing a pointer. In this way the references to other nodes from a node inside this buffer may be represented by an integer offset from the start of the buffer. Every internal state is represented by a pointer to its root node inside this buffer; this pointer can be also represented by an integer offset from the start of the buffer.

3.2.3. The internal state of a catchain is flushed to an append-only file. The consequence of the structure of the buffer used to store the internal states of a catchain explained above is that it is updated only by appending some new data at its end. This means that the internal state (or rather the buffer containing all the required internal states) of a catchain can be flushed to an append-only file, and easily recovered after a restart. The only other data that needs to be stored before restarts is the offset (from the start of the buffer, i.e., of this file) of the current state of the catchain. A simple key-value database can be used for this purpose.

3.2.4. Sharing data between different states. It turns out that the tree (or rather the dag) representing the new state $\sigma_{S\cup\{m\}} = f(\sigma_S, m)$ shares large subtrees with the previous state σ_S, and, similarly, $\sigma_{S\cup T} = g(\sigma_S, \sigma_T)$ shares large subtrees with σ_S and σ_T. The persistent structure used for representing the states in BCP makes it possible to reuse the same pointers inside the buffer for representing such shared data structures instead of duplicating them.

3.2.5. Memoizing nodes. Another technique employed while computing new states (i.e., the values of function f) is that of *memoizing new nodes*, also borrowed from functional programming languages. Namely, whenever a new node is constructed (inside the huge buffer containing all states for a specific catchain), its hash is computed, and a simple hash table is used to look up the latest node with the same hash. If a node with this hash is found, and it has the same contents, then the newly-constructed node is discarded and a reference to the old node with the same contents is returned instead. On the other hand, if no copy of the new node is found, then the hash table is updated, the end-of-buffer (allocation) pointer is advanced, and the pointer to the new node is returned to the caller.

In this way if different processes end up making similar computations and having similar states, large portions of these states will be shared even if they are not directly related by application of function f as explained in **3.2.4**.

3.2.6. Importance of optimization techniques. The optimization techniques **3.2.4** and **3.2.5** used for sharing parts of different internal states inside the same catchain are drastically important for improving the memory profile and the performance of BCM in a large process group. The improvement is several orders of magnitude in groups of $N \approx 100$ processes. Without these optimizations BCM would not be fit for its intended purpose (BFT consensus on new blocks generated by validators in the TON Blockchain).

3.2.7. Message m contains a hash of state $\sigma_{D_m^+}$. Every message m contains a (Merkle) hash of (the abstract representation of) the corresponding state $\sigma_{D_m^+}$. Very roughly, this hash is computed recursively using the tree of nodes representation of **3.2.1**: all node references inside a node are replaced with (recursively computed) hashes of the referred nodes, and a simple 64-bit hash of the resulting byte sequence is computed. This hash is also used for memoization as described in **3.2.5**.

The purpose of this field in messages is to provide a sanity check for the

computations of $\sigma_{D_m^+}$ performed by different processes (and possibly by different implementations of the state update algorithm): once $\sigma_{D_m^+}$ is computed for a newly-delivered message m, the hash of computed $\sigma_{D_m^+}$ is compared to the value stored in the header of m. If these values are not equal, an error message is output into an error log (and no further actions are taken by the software). These error logs can be examined to detect bugs or incompatibilities between different versions of BCP.

3.3. State recovery after restart or crashes. A catchain is typically used by the BCP for several minutes; during this period, the program (the validator software) running the Catchain protocol may be terminated and restarted, either deliberately (e.g., because of a scheduled software update) or unintentionally (the program might crash because of a bug in this or some other subsystem, and be restarted afterwards). One way of dealing with this situation would be to ignore all catchains not created after the last restart. However, this would lead to some validators not participating in creating any blocks for several minutes (until the next catchain instances are created), which is undesirable. Therefore, a catchain state recovery protocol is run instead after every restart, so that the validator can continue participating in the same catchain.

3.3.1. Database of all delivered messages. To this end, a special database is created for each active catchain. This database contains all known and delivered messages, indexed by their identifiers (hashes). A simple key-value database suffices for this purpose. The hash of the most recent outbound message $m = m_{i,s}$ generated by the current process i is also stored in this database. After restart, all messages up to m are recursively delivered in proper order (in the same way as if all these messages had been just received from the network in an arbitrary order) and processed by the higher-level protocol, until m finally is delivered, thus recovering the current state.

3.3.2. Flushing new messages to disk. We have already explained in **2.6** that newly-created messages are stored in the database of all delivered messages (cf. **3.3.1**) and the database is flushed to disk before the new message is sent to all network neighbors. In this way we can be sure that the message cannot be lost if the system crashes and is restarted, thus avoiding the creation of involuntary forks.

3.3.3. Avoiding the recomputation of states $\sigma_{D_m^+}$. An implementation might use an append-only file containing all previously computed states as described in **3.2.3** to avoid recomputing all states after restart, trading off disk space for computational power. However, the current implementation does not use this optimization.

3.4. High-level description of Block Consensus Protocol. Now we are ready to present a high-level description of the Block Consensus Protocol employed by TON Blockchain validators to generate and achieve consensus on new blockchain blocks. Essentially, it is a three-phase commit protocol that runs over a catchain (an instance of the Catchain protocol), which is used as a "hardened" message broadcast system in a process group.

3.4.1. Creation of new catchain messages. Recall that the lower-level Catchain protocol does not create broadcast messages on its own (with the only exception being service broadcasts with fork proofs, cf. **2.7.5**). Instead, when a new message needs to be created, the higher-level protocol (BCP) is asked to do this by invoking a callback. Apart from that, the creation of new messages may be triggered by changes in the current virtual state and by timer alarms.

3.4.2. Payload of catchain messages. In this way the payload of catchain messages is always determined by the higher level protocol, such as BCP. For BCP, this payload consists of

- Current Unix time. It must be non-decreasing on subsequent messages of the same process. (If this restriction is violated, all processes processing this message will tacitly replace this Unix time by the maximum Unix time seen in previous messages of the same sender.)
- Several (zero or more) *BCP events* of one of the admissible types listed below.

3.4.3. BCP events. We have just explained that the payload of a catchain message contains several (possibly zero) BCP events. Now we list all admissible BCP event types.

- Submit(*round*, *candidate*) — suggest a new block candidate
- Approve(*round*, *candidate*, *signature*) — a block candidate has passed local validation

- Reject(*round*, *candidate*) — a block candidate has failed local validation
- CommitSign(*round*, *candidate*, *signature*) — a block candidate has been accepted and signed
- Vote(*round*, *candidate*) — a vote for a block candidate
- VoteFor(*round*, *candidate*) — this block candidate must be voted for in this round (even if the current process has another opinion)
- PreCommit(*round*, *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
- Δ_i for $1 \leq i \leq C$ — delay before suggesting the block candidate with priority i
- Δ_∞ — 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*.[6]

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 block producers is computed by a deterministic algorithm (in the simplest case, processes i, $i+1$, ..., $i+C-1$ 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

[6]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.

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 Δ_i, where i is the index of the producer in the list of designated block producers, with $0 = \Delta_1 \leq \Delta_2 \leq \ldots$. After some predetermined period of time Δ_∞ 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 Δ_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 σ_{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 σ_{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 Δ_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 Δ_∞ 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 PreCommited 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).[7] 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).
- 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).

[7] 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 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 τ, where τ 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, ... 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 Δ_∞ 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.