Telegram Open Network

Nikolai Durov
TL: Dr Awesome Doge

August 12, 2023

Abstract

本論文旨在首次詳述 Telegram Open Network (TON) 以及相關的區塊鏈、P2P (點對點)、分散式儲存及服務託管技術。為了確保本文件的篇幅在合理的範疇內，我們主要針對 TON 平台中具有獨特性和關鍵定義的功能進行探討，這些功能對於實現其明確設定的目標具有核心重要性。

緒論

Telegram Open Network (TON) 是一個快速、安全且可擴展的區塊鏈和網絡專案，如有必要，它能夠處理每秒數百萬筆交易，對使用者和服務提供者都非常友好。我們希望它能夠承載所有目前提議和構想的合理應用程式。人們可以將 TON 視為一個巨大的分散式超級電腦，或者更恰當地說，一個巨大的 ‘‘超級伺服器’’，旨在托管和提供各種服務。

本文並非關於所有實作細節的終極參考。在開發和測試階段，部分具體內容有可能發生變更。

Contents

1 TON 組件的簡要描述

Telegram Open Network (TON) 結合了以下組件：

- 一個靈活的多區塊鏈平台 (*TON Blockchain*; 參考第 **2**章)，能夠處理每秒數百萬筆交易，擁有圖靈完全智能合約、可升級的正式區塊鏈規格、多加密貨幣價值轉移、支持微支付通道和鏈下支付網絡。*TON Blockchain* 提供了一些新的和獨特的功能，例如「自我修復」垂直區塊鏈機制 (參考 **2.1.17**) 和 Instant Hypercube Routing (參考 **2.4.20**)，使其同時快速、可靠、可擴展和自我一致。

- 一個點對點網絡 (*TON P2P Network*, 或簡稱 *TON Network*; 參考第 **4**章)，用於訪問 TON 區塊鏈、發送交易候選者，以及只接收客戶端感興趣的區塊鏈的部分更新（例如，與客戶端的賬戶和智能合約相關的部分），但也能支持任意分散服務，無論是否與區塊鏈相關。

- 一種分散的文件存儲技術 *(TON Storage);* (參考 **5.1.8**)，通過 *TON Network* 訪問，由 TON Blockchain 用於存儲區塊和狀態數據（快照）的存檔副本，但也可以用於存儲平台上的使用者或其他服務的任意文件，使用類似於 torrent 的訪問技術。

- 一個網絡代理/匿名層 *(TON Proxy);* (參考 **5.1.11** 和 **4.0.6**)，類似於 I^2P (Invisible Internet Project)，如有必要（例如，從擁有大量加密貨幣的賬戶提交交易的節點，或希望隱藏其確切 IP 地址和地理位置以對抗 DDoS 攻擊的高風險區塊鏈驗證器節點），用於隱藏 *TON Network* 節點的身份和 IP 地址。

- 一個類似 Kademlia 的分散式哈希表 (*TON DHT*; 參考 **4.1**)，用作 *TON Storage* 的「torrent tracker」(參考 **4.1.10**)，作為 *TON Proxy* 的「input tunnel locator」(參考 **4.1.14**)，以及作為 *TON Services* 的服務定位器 (參考 **4.1.12**)。

- 一個提供任意服務的平台 (*TON Services*; 參考第 **5**章)，居住在並可通過 *TON Network* 和 *TON Proxy* 訪問，具有正式化的界面 (參考 **5.3.14**) 使得瀏覽器或智能手機應用程序可以互動。這些正式界面和持久的服務入口點可以在 TON Blockchain 中發布 (參考 **5.3.17**)；提供服務的實際節點可以從在 TON Blockchain 中發布的信息開始，通過 *TON DHT* 查找 (參考 **4.1.12**)。服務可以在 TON Blockchain 中創建智能合約，為其客戶提供一些保證 (參考 **5.1.7**)。

- *TON DNS* (參考 **5.3.1**)，用於為賬戶、智能合約、服務和網絡節點分配易讀的名稱。
- *TON Payments* (參考第 **6**章)，一個用於微支付、微支付通道和微支付通道網絡的平台。它可用於快速的鏈下價值轉移，以及支付由 *TON Services* 提供的服務。
- TON 將允許輕鬆集成第三方消息和社交網絡應用程序，從而使區塊鏈技術和分散服務終於可用且可被普通使用者訪問 (參考 **5.3.24**)，而不僅僅是少數早期的加密貨幣採用者。我們將在我們的另一個項目中，Telegram Messenger (參考 **5.3.19**)，提供這樣的集成例子。

雖然 TON Blockchain 是 TON 專案的核心，而其他組件可能被視為對區塊鏈扮演輔助角色，但它們自身也具有有趣和實用的功能。結合使用，它們允許平台容納比僅使用 TON Blockchain 更多樣化的應用程式 (參考 **2.9.13** 和 **5.1**)。

2 TON Blockchain

我們從描述 Telegram Open Network (TON) Blockchain 開始，這是該專案的核心組件。我們這裡的方法是「由上而下」：我們首先給出整體的一般描述，然後提供每個組件的更多細節。

為了簡單起見，我們在此談論 *the/* TON Blockchain，即使原則上這種區塊鏈協議可能有多個獨立運行的實例（例如，由於硬分叉的結果）。我們只考慮其中之一。

2.1 TON Blockchain 作為 2-區塊鏈的集合

TON Blockchain 實際上是區塊鏈的集合（甚至是區塊鏈的區塊鏈集合，或稱為 *2*-區塊鏈---這一點將在 **2.1.17**中進一步說明），因為沒有單一的區塊鏈專案能夠達到我們每秒處理數百萬交易的目標，而不是現在的每秒數十次交易的標準。

2.1.1. List of blockchain types. 此系列中的區塊鏈包括：

- 唯一的 *master blockchain*，或簡稱為 *masterchain*，該區塊鏈包含有關協議的一般資訊、其參數的當前值、驗證者集合和他們的股份、當前活躍的工作鏈 (workchains) 及其"shards"，以及最重要的，所有 workchains 和 shardchains 的最近區塊的哈希集合。

- 數個 (最多 2^{32}) 的 *working blockchains*，或簡稱為 *workchains*，實際上是這系統的" 工作馬"，包含價值轉移和智慧合約交易。不同的 workchains 可能有不同的" 規則"，意味著帳戶地址的不同格式、交易的不同格式、智慧合約的不同虛擬機 (VMs)、不同的基本加密貨幣等等。然而，它們都必須滿足某些基本的互操作性標準，以確保不同的 workchains 之間的互動簡單和可能。在這方面，TON Blockchain 是 *heterogeneous*（參考 **2.8.8**），類似於 EOS（參考 **2.9.7**）和 PolkaDot（參考 **2.9.8**）項目。

- 每個 workchain 會進一步細分為多達 2^{60} 的 *shard blockchains*，或簡稱為 *shardchains*，它們具有與 workchain 本身相同的規則和區塊格式，但只對帳戶的某個子集負責，這取決於帳戶地址的幾個首位 (最重要的位)。換句話說，這系統內建了一種分片 (sharding) 的形式（參考 **2.8.12**）。因為所有這些 shardchains 共享通用的區塊格式和規則，TON Blockchain 在這方面是 *homogeneous*（參考 **2.8.8**），這與 Ethereum 的某個擴展建議相似。[1]

[1] `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`

- shardchain（和 masterchain）中的每個區塊實際上不只是一個區塊，而是一個小區塊鏈。通常，這"block blockchain" 或"vertical blockchain" 只包含一個區塊，然後我們可能會認為這只是 shardchain 的相對應區塊（在這種情況下也稱為"horizontal blockchain"）。但是，如果需要修正不正確的 shardchain 區塊，新的區塊將被提交到"vertical blockchain"，包含無效"horizontal blockchain" 區塊的替代品，或一個"block difference"，只包含該區塊先前版本中需要更改的部分的描述。這是一個 TON 特有的機制，用於替換檢測到的無效區塊，而不會真正地分叉所有涉及的 shardchains；這將在 **2.1.17**中詳細解釋。目前，我們只需指出，每個 shardchain（和 masterchain）不是一個常規的區塊鏈，而是一個 *blockchain of blockchains*，或 *2D-blockchain*，或只是一個 *2-blockchain*。

2.1.2. Infinite Sharding Paradigm. 無限分片範式。幾乎所有的區塊鏈分片提案都是「自上而下」：首先想像一個單一的區塊鏈，然後討論如何將它分割成幾個互動的分片鏈以提高效能和達到可擴展性。

TON 的分片方法是「自下而上」，如下所述。

想像分片已被極端化，以至於每個分片鏈中只剩下一個帳戶或智能合約。然後我們有大量的「帳戶鏈」，每個鏈描述只有一個帳戶的狀態和狀態過渡，並向彼此發送具有價值的消息以傳輸價值和信息。

當然，擁有數億的區塊鏈是不切實際的，每個鏈中的更新（即新的區塊）通常出現得相對較少。為了更有效地實施它們，我們將這些「帳戶鏈」組合成「分片鏈」，以便分片鏈的每個區塊基本上是已分配給此分片的帳戶鏈的區塊的集合。因此，「帳戶鏈」只在「分片鏈」內部擁有純粹的虛擬或邏輯存在。

我們稱這種觀點為「無限分片範式」。它解釋了 TON 區塊鏈的許多設計決策。

2.1.3. Messages. Instant Hypercube Routing. 消息。即時超立方路由。無限分片範式告訴我們將每個帳戶（或智能合約）視為它自己的分片鏈中。然後，一個帳戶可能影響另一帳戶的狀態的唯一方式是向它發送一個「消息」（這是所謂的 Actor 模型的特殊實例，其中帳戶作為 Actors；cf. **2.4.2**）。因此，帳戶間（和分片鏈間，因為源帳戶和目的地帳戶，一般來說，位於不同的分片鏈中）的消息系統對於像 TON 區塊鏈這樣的可擴展系統非常重要。事實上，TON 區塊鏈的一個新特性，稱為「即時超立方路由」（cf. **2.4.20**），使它能夠將消息從一個分片鏈的區塊傳遞和處理到目的地分片鏈的下一個區塊，不考慮系統中的分片鏈總數。

2.1.4. Quantity of masterchains, workchains and shardchains. TON

區塊鏈中恰有一個主鏈 (masterchain)。但是，此系統有潛能容納高達 2^{32} 的工作鏈 (workchains)，每個工作鏈都可細分為高達 2^{60} 的分片鏈 (shardchains)。

2.1.5. Workchains can be virtual blockchains, not true blockchains. 由於工作鏈通常被細分為分片鏈，工作鏈的存在是「虛擬的」，這意味著它不是一個真正的區塊鏈，如 **2.2.1**下提供的一般定義所描述，而只是一組分片鏈的集合。當只有一個分片鏈對應到一個工作鏈時，這個獨特的分片鏈可能與工作鏈相同，這樣它在某個時間點變成一個「真正的」區塊鏈，進而與常規的單一區塊鏈設計有相似性。然而，無限分片範式 (cf. **2.1.2**) 告訴我們這種相似性確實是表面的：能夠將潛在的大量「帳戶鏈」暫時分組到一個區塊鏈只是一個巧合。

2.1.6. Identification of workchains. 每一個工作鏈都由其 *number* 或 *workchain identifier*（$workchain\_id : uint_{32}$）來識別，它只是一個無符號的 32 位整數。工作鏈是由主鏈中的特殊交易所創建，定義（先前未使用的）工作鏈識別符和工作鏈的正式描述，至少足以讓此工作鏈與其他工作鏈互動以及對此工作鏈的區塊進行表面驗證。

2.1.7. Creation and activation of new workchains. 新的工作鏈的創建可以由社區中的任何成員啟動，只要他們準備支付發佈新工作鏈的正式規範所需的（高額）主鏈交易費用。但是，為了使新的工作鏈變得活躍，需要三分之二的驗證者達成共識，因為他們需要升級他們的軟件以處理新工作鏈的區塊，並通過特殊的主鏈交易表示他們準備好與新的工作鏈一起工作。對新工作鏈的啟動感興趣的方可能會提供某些激勵，讓驗證者透過智能合約分發的某些獎勵來支持新的工作鏈。

2.1.8. Identification of shardchains. 每個分片鏈 (shardchain) 都由一對 $(w, s) = (workchain\_id, shard\_prefix)$ 來識別，其中 $workchain\_id : uint_{32}$ 識別相應的工作鏈 (workchain)，而 $shard\_prefix : \mathbf{2}^{0\ldots60}$ 是一個最長為 60 的位元串，定義此分片鏈所負責的帳戶子集。換句話說，所有以 *shard_prefix* 開頭的帳戶 *account_id* （即，具有 *shard_prefix* 作為最重要位元）都將被分配到這個分片鏈。

2.1.9. Identification of account-chains. 回憶一下，帳戶鏈 (account-chains) 只有虛擬存在 (cf. **2.1.2**)。然而，它們有一個自然的識別符——即，$(workchain\_id, account\_id)$——因為任何帳戶鏈都包含關於恰好一個帳戶（無論是簡單帳戶還是智慧合約——這裡的區別不重要）的狀態和更新的信息。

2.1.10. Dynamic splitting and merging of shardchains; cf. 2.7. 一個較不複雜的系統可能使用 *static sharding*，例如，使用 *account_id* 的前八位來選擇 256 個預定義的碎片之一。

TON 區塊鏈的一個重要特點是它實現了 *dynamic sharding*，這意味著碎片的數量不是固定的。相反，如果滿足某些正式條件（基本上，如果原始碎片上的交易負載在很長的時間內都足夠高），分片 (w, s) 可以自動細分為分片 $(w, s.0)$ 和 $(w, s.1)$。相反，如果負載在一段時間內保持得太低，分片 $(w, s.0)$ 和 $(w, s.1)$ 可以自動合併回分片 (w, s)。

最初，只為工作鏈 w 創建了一個分片 $(w, \emptyset)$。稍後，當這變得必要時，它被細分為更多的分片 (cf. **2.7.6** and **2.7.8**)。

2.1.11. Basic workchain or Workchain Zero. 雖然可以定義高達 2^{32} 的工作鏈 (workchains) 並有其特定的規則和交易，但我們一開始只定義一個，即 $workchain\_id = 0$。這個工作鏈被稱為 Workchain Zero 或基礎工作鏈，它是用於操作 *TON smart contracts* 和轉移 *TON coins*，也稱為 *Grams* (cf. Appendix A)。大多數應用可能只需要 Workchain Zero。基礎工作鏈的分片鏈會被稱為 *basic shardchains*。

2.1.12. Block generation intervals. 我們預計每個分片鏈和主鏈大約每五秒會生成一個新的區塊。這將導致相對較小的交易確認時間。所有分片鏈的新區塊大約同時生成；主鏈的新區塊大約在一秒後生成，因為它必須包含所有分片鏈的最新區塊的雜湊值。

2.1.13. Using the masterchain to make workchains and shardchains tightly coupled. 一旦分片鏈的區塊的雜湊值被合併到主鏈的區塊中，該分片鏈區塊及其所有祖先都被認為是「正規的」，這意味著它們可以被所有分片鏈的後續區塊引用為固定且不可變的內容。實際上，每個新的分片鏈區塊都包含最近的主鏈區塊的雜湊值，並且從該主鏈區塊引用的所有分片鏈區塊在新區塊中都被認為是不可變的。

從本質上講，這意味著在分片鏈區塊中提交的交易或消息可以在其他分片鏈的下一個區塊中安全地使用，而不需要等待，例如，二十次確認（即在同一區塊鏈中在原始區塊之後生成的二十個區塊）之前轉發消息或基於之前的交易採取其他操作，這在大多數建議的「鬆散連接」系統中很常見 (cf. **2.8.14**)，如 EOS。我們相信，這種能力在提交後的五秒內在其他分片鏈中使用交易和消息是我們這種「緊密連接」系統能夠提供前所未有的性能的原因之一 (cf. **2.8.12** and **2.8.14**)。

2.1.14. Masterchain block hash as a global state. 根據 **2.1.13**，最後一個主鏈區塊的雜湊完全確定了外部觀察者的整體系統狀態。人們不需要單獨監視所有分片鏈的狀態。

2.1.15. Generation of new blocks by validators; cf. 2.6. TON 區塊鏈使用 Proof-of-Stake (PoS) 方法在分片鏈和主鏈中生成新的區塊。這意味著有一組，例如，多達幾百個的 *validators*---特殊的節點，透過特殊的主鏈交易存放 *stakes*（大量的 TON coins）以符合生成和驗證新區塊的資格。

然後，小部分的驗證器被分配給每一個分片 (w, s)，這是以決定性的偽隨機方式進行的，每 1024 個區塊大約會改變一次。這部分的驗證器建議並在下一個分片鏈區塊應是什麼上達成共識，通過從客戶端收集適當的建議交易到新的有效的區塊候選。對於每個區塊，驗證器上有一個偽隨機選定的順序，以確定誰的區塊候選在每一輪中有最高的優先順序被提交。

驗證器和其他節點檢查所提議的區塊候選的有效性；如果驗證器簽署了一個無效的區塊候選，它可能會自動被懲罰，失去部分或全部的 stake，或被暫停從驗證器集合一段時間。之後，驗證器應達成對下一個區塊的選擇的共識，本質上是 BFT (Byzantine Fault Tolerant; cf. **2.8.4**) 共識協議的高效變體，類似於 PBFT [4] 或 Honey Badger BFT [11]。如果達到共識，將創建新的區塊，且驗證器在交易費用上進行分割，包括交易，加上一些新創建的（“鑄造的”）硬幣。

每個驗證器可以被選舉參與多個驗證器子集；在這種情況下，預計它將平行運行所有的驗證和共識算法。

在生成所有新的分片鏈區塊之後或超時之後，將生成一個新的主鏈區塊，包括所有分片鏈的最新區塊的雜湊值。這是通過 *all* 驗證器的 BFT 共識完成的。[2]

TON PoS 方法及其經濟模型的更多細節提供在 section **2.6**。

2.1.16. Forks of the masterchain. 我們緊密耦合的方法帶來的一個複雜性是，切換到主鏈的不同分叉幾乎必然需要在至少一些分片鏈中切換到另一個分叉。另一方面，只要主鏈中沒有分叉，分片鏈中的分叉甚至都是不可能的，因為分片鏈的替代分叉中的沒有區塊可以通過將其哈希值納入主鏈區塊而變得「標準化」。

一般的規則是，如果主鏈區塊 B' 是 B 的前輩，B' 包括 (w, s)-分片鏈區塊 $B'_{w,s}$ 的哈希 $\mathit{HASH}(B'_{w,s})$，且 B 包括哈希 $\mathit{HASH}(B_{w,s})$，那麼 $B'_{w,s}$ 必須是 $B_{w,s}$ 的前輩*;* 否則，主鏈區塊 B 就是無效的。

我們預期主鏈分叉將會很少，幾乎不存在，因為在由 TON 區塊鏈所採用的 BFT 範疇中，它們只能在大部分 驗證器行為不正確的情況下發生（參見 **2.6.1** 和 **2.6.15**），這將意味著由違反者承擔的重大的 stake 損失。因此，不應期待分片鏈中存在真正的分叉。相反，如果檢測到一個無效的分片鏈區塊，將通過 2-blockchain 的「垂直區塊鏈」機制進行修正（參見 **2.1.17**），

[2] 實際上，兩個三分之一的 stake 就足夠達成共識，但努力收集盡可能多的簽名。

這可以在不分叉「水平區塊鏈」（即，分片鏈）的情況下實現此目標。同樣的機制也可以用來修正主鏈區塊中的非致命性錯誤。

2.1.17. Correcting invalid shardchain blocks. 通常，只有有效的分片鏈區塊會被提交，因為分配給分片鏈的驗證器在新區塊可以被提交之前必須達到三分之二的拜占庭共識。然而，系統必須允許檢測先前提交的無效區塊及其校正。

當然，一旦找到一個無效的分片鏈區塊——不論是由一個驗證器（不一定分配到這個分片鏈）還是一個「漁夫」（系統的任何節點，它已經支付了某個存款以便對區塊有效性提出疑問；參見 **2.6.4**）——無效性的主張及其證明都會被提交到主鏈，而已經簽署無效區塊的驗證器將被懲罰，部分或全部扣除他們的 stake，和/或暫時從驗證器集合中被停權（後者的措施對於攻擊者竊取本質上良性的驗證器的私有簽名鍵非常重要）。

然而，這還不夠，因為由於先前提交的無效分片鏈區塊，系統（TON 區塊鏈）的整體狀態結果是無效的。這個無效區塊必須被一個新的有效版本替換。

大多數系統會通過「回滾」到該分片鏈中的無效區塊之前的最後一個區塊，以及每個其他分片鏈中不受從無效區塊傳播的消息影響的最後區塊，並從這些區塊創建一個新的分叉來實現這一點。這種方法的缺點是，大量本來正確且已提交的交易突然被回滾，且不清楚它們是否會在稍後再次被包括。

TON 區塊鏈通過使每個分片鏈和主鏈的每個「區塊」（「水平區塊鏈」）本身都成為一個小型區塊鏈（「垂直區塊鏈」），包含這個「區塊」的不同版本，或其「差異」來解決這個問題。通常，垂直區塊鏈只包含一個區塊，而分片鏈看起來像一個經典的區塊鏈。但是，一旦一個區塊的無效性被確認並提交到主鏈區塊中，該無效區塊的「垂直區塊鏈」就被允許在垂直方向上增加一個新區塊，替換或編輯無效區塊。這個新區塊是由當前問題分片鏈的驗證器子集生成的。

新的“垂直”區塊要有效的規則非常嚴格。特別是，如果無效區塊中包含的虛擬“帳戶鏈區塊”（參見 **2.1.2**）本身是有效的，則它必須被新的垂直區塊保持不變。

一旦在無效區塊上方提交了新的“垂直”區塊，它的哈希就會在新的 Masterchain 區塊中公布（或者更正確地說，在原始 Masterchain 區塊上方的新“垂直”區塊中公布，該區塊中最初發布了無效 Shardchain 區塊的哈希），並且進一步將更改傳播到任何參照此區塊的 Shardchain 區塊（例如，那些從不正確的區塊接收消息的區塊）。這可以通過在先前參照“不正確”區塊的所有區塊的垂直區塊鏈中提交新的“垂直”區塊來進行修正；新的垂直區塊將參照最新（已更正）的版本。同樣，嚴格的規則禁止更改未受

到實際影響的帳戶鏈（即，與前一版本中收到的消息相同的帳戶鏈）。通過這種方式，修正不正確的區塊產生“漣漪”，最終傳播到所有受影響的 Shardchain 的最新區塊；這些更改也反映在新的“垂直”Masterchain 區塊中。

一旦“歷史重寫”的漣漪到達最新區塊，新的 Shardchain 區塊僅以一個版本生成，僅作為最新區塊版本的後繼者。這意味著它們將從一開始就包含對正確（最新）的垂直區塊的引用。

Masterchain 狀態隱含地定義了一個映射，將每個“垂直”區塊鏈的第一個區塊的哈希轉換為其最新版本的哈希。這使客戶端可以通過其第一個（通常是唯一的）區塊的哈希識別和定位任何垂直區塊鏈。

2.1.18. TON coins and multi-currency workchains. TON 區塊鏈支持多達 2^{32} 種不同的“加密貨幣”、“硬幣”或“代幣”，通過 32 位的 *currency_id* 加以區分。新的加密貨幣可以通過主鏈中的特殊交易來添加。每個工作鏈都有一種基本加密貨幣，並且可以有幾種其他加密貨幣。

有一種特殊的加密貨幣，*currency_id* = 0，即 *TON* 幣，也稱為 *Gram*（參見附錄 A）。這是工作鏈零的基本加密貨幣。它也用於交易費和驗證者的權益股份。

原則上，其他工作鏈可能會以其他代幣收取交易費。在這種情況下，應該提供一些智能合約，用於將這些交易費自動轉換為 Grams。

2.1.19. Messaging and value transfer. 屬於相同或不同工作鏈的分片鏈可以互相傳送 *messages*。儘管允許的訊息的確切形式取決於接收工作鏈和接收帳戶（智慧合約），但有一些共同的欄位使跨工作鏈的訊息成為可能。特別是，每條訊息可能會有一些 *value*，以一定數量的 Grams（TON coins）和/或其他註冊的加密貨幣的形式附加，前提是它們被接收工作鏈宣布為可接受的加密貨幣。

這種訊息的最簡單形式是從一個（通常不是智慧合約）帳戶到另一個帳戶的價值轉移。

2.1.20. TON Virtual Machine. *TON Virtual Machine*，也縮寫為 *TON VM* 或 *TVM*，是用於在主鏈和基本工作鏈中執行智慧合約代碼的虛擬機器。其他工作鏈可能使用其他虛擬機器，與 TVM 一起或替代 TVM。

以下我們列出了其一些特性。它們在 **2.3.12**、**2.3.14**和其他地方進一步討論。

- TVM 將所有資料表示為一系列的 *(TVM) cells*/（參考 **2.3.14**）。每一個 cell 包含最多 128 個資料位元組，以及最多 4 個指向其他 cells 的參考。基於“一切皆為 cell 包”的理念 (參考 **2.5.14**)，這使得 TVM

能夠處理與 TON 區塊鏈相關的所有資料，包括必要時的區塊和區塊鏈全局狀態。

- TVM 可以處理任意代數資料型態的值 (參考 **2.3.12**)，表示為 TVM cells 的樹或有向非循環圖。但它對代數資料型態的存在是不知情的；它只是處理 cells。
- TVM 內建支援 hashmaps (參考 **2.3.7**)。
- TVM 是一個堆疊機器。它的堆疊保存 64 位元整數或 cell 參考。
- 支援 64-bit, 128-bit 和 256-bit 的算術運算。所有 n-bit 的算術操作都有三種形式：用於無符號整數，用於有符號整數，以及對於模 2^n 的整數（後者不自動檢查溢出）。
- TVM 有從 n-bit 轉換到 m-bit 的無符號和有符號整數，對所有 $0 \leq m, n \leq 256$，並具有溢出檢查。
- 所有算術操作預設執行溢出檢查，這大大簡化了智能合約的開發。
- TVM 有“乘然後移位”和“移位然後除”的算術操作，中間值在更大的整數型態中計算；這簡化了固定點算術的實現。
- TVM 提供對位串和字節串的支援。
- 提供對某些預定義曲線，包括 Curve25519 的 256-bit 橢圓曲線加密 (ECC) 的支援。
- 對於在某些橢圓曲線上的 Weil 配對的支援也存在，這對於快速實現 zk-SNARKs 很有用。
- 支援包括 sha256 在內的流行雜湊函數。
- TVM 可以處理 Merkle 證明 (參考 **6.1.9**)。
- TVM 提供對“大型”或“全局”智能合約的支援。這類智能合約必須知道分片 (參考 **2.3.18** 和 **2.3.16**)。常規（本地）智能合約可以是不知道分片的。
- TVM 支援閉包。
- 一個“無脊標籤 G-機器”[13] 可以容易地在 TVM 內部實現。

除了“TVM 組件”，還可以為 TVM 設計幾種高級語言。所有這些語言都將具有靜態類型，並支援代數資料型態。我們預見以下可能性：

- 類似 Java 的命令式語言，每個智能合約都像一個獨立的類。
- 惰性的功能語言 (如 Haskell)。
- 積極的功能語言 (如 ML)。

2.1.21. Configurable parameters. TON Blockchain 的一個重要特性是它有許多可配置的參數。這意味著它們是 masterchain 狀態的一部分，且可以通過 masterchain 中的某些特殊的提議/投票/結果交易來更改，而不需要硬分叉。更改這些參數需要收集三分之二的驗證者投票，以及超過一半想要參與投票過程的所有其他參與者的投票以支持該提議。

2.2 關於區塊鏈的概論

2.2.1. General blockchain definition. 一般來說，任何 *(*真實的*)* 區塊鏈都是一系列的區塊，每個區塊 B 都包含一個參考至前一個區塊的 $\text{BLK-PREV}(B)$（通常是將前一個區塊的雜湊包含在當前區塊的標頭中），以及一個交易的列表。每筆交易都描述了全球區塊鏈狀態的某種轉換; 一個區塊中列出的交易是按順序應用的，從舊狀態開始計算新狀態，這是在評估前一個區塊後的結果狀態。

2.2.2. Relevance for the TON Blockchain. 請注意，*TON Blockchain* 並不是真正的區塊鏈，而是 2-區塊鏈的集合（即，區塊鏈的區塊鏈集合; 參考 **2.1.1**），所以上述內容並不直接適用於它。然而，我們從真正的區塊鏈的這些一般性開始，用它們作為我們更複雜建構的基礎。

2.2.3. Blockchain instance and blockchain type. 人們經常使用「*blockchain*」一詞來表示一般的 *blockchain type* 和其特定的 *blockchain instances*，定義為滿足某些條件的區塊序列。例如，**2.2.1** 是指區塊鏈的實例。

由此，一個區塊鏈類型通常是類型 $Block^*$ 的「子類型」，由那些滿足某些相容性和有效性條件的區塊序列組成：

$$Blockchain \subset Block^* \tag{1}$$

更好的定義方式是說 *Blockchain* 是一個 *dependent couple type*，由 couple $(\mathbb{B}, v)$ 組成，第一部分 $\mathbb{B} : Block^*$ 是類型 $Block^*$（即區塊列表），第二部分 $v : isValidBc(\mathbb{B})$ 是 $\mathbb{B}$ 的有效性的證明或證人。由此，

$$Blockchain \equiv \Sigma_{(\mathbb{B}:Block^*)} isValidBc(\mathbb{B}) \tag{2}$$

我們在這裡使用了從 [16] 借用的依賴型態和的表示法。

2.2.4. Dependent type theory, Coq and TL. 注意，我們在此使用的是（Martin-Löf）依賴型態理論，類似於 Coq[3] 證明助手中使用的。依賴型態理論的簡化版本也用於 *TL (Type Language)*[4]，將在 TON Blockchain 的正式規範中使用，描述所有資料結構的序列化以及區塊、交易等的佈局。

事實上，依賴型態理論提供了對證明是什麼的有用形式化。當需要提供某個區塊的無效性證明時，這種形式的證明（或它們的序列化）可能會變得很有用。

2.2.5. TL, or the Type Language. 由於 TL（Type Language）將被用於 TON 區塊、交易和網路數據包的正式規範，因此值得簡短地討論。

TL 是一種適用於描述依賴代數的 *types* 的語言，允許具有數字（自然數）和型態參數。每個型態都通過幾個 *constructors* 來描述。每個構造器都有一個（人類可讀的）識別碼和一個 *name*，這是一個位元組列（預設為 32 位整數）。除此之外，構造器的定義包含一個與其型態一起的字段列表。

構造器和型態定義的集合被稱為 *TL-scheme*。它通常保存在一個或多個帶有 `.tl` 後綴的文件中。

TL-schemes 的一個重要特性是，它們確定了序列化和反序列化定義的代數型態的值（或對象）的明確方式。具體來說，當一個值需要被序列化為一串位元組時，首先序列化用於此值的構造器的名稱。然後是每個字段的遞迴計算的序列化。

一個適合序列化任意對象為 32 位整數序列的 TL 的先前版本的描述，可以在 `https://core.telegram.org/mtproto/TL` 上找到。一個新版本的 TL，名為 *TL-B*，正在開發中，用於描述 TON Project 使用的對象的序列化。這個新版本可以將對象序列化為位元組流，甚至是位元流（而不僅僅是 32 位整數），並提供將其序列化為 TVM cell 樹的支持（cf. **2.3.14**）。TL-B 的描述將是 TON Blockchain 的正式規範的一部分。

2.2.6. Blocks and transactions as state transformation operators. 通常，任何區塊鏈 (型態) *Blockchain* 都有一個關聯的全域狀態 (型態) *State*，以及一個交易 (型態) *Transaction*。區塊鏈的語意在很大程度上是由交易應用函數所決定的：

$$ev\_trans' : Transaction \times State \to State^{?} \tag{3}$$

這裡的 $X^{?}$ 表示 $\textsc{Maybe}\, X$，是將 Maybe 單子應用於型態 X 的結果。這與我們使用 X^{*} 表示 $\textsc{List}\, X$ 類似。本質上，型態 $X^{?}$ 的值要麼是型態 X

[3] `https://coq.inria.fr`

[4] `https://core.telegram.org/mtproto/TL`

的值，要麼是一個特殊值 $\perp$ 表示實際值的缺失 (想想空指標)。在我們的情境中，我們使用 $State^?$ 而不是 $State$ 作為結果型態，因為某個交易從某些原始狀態呼叫可能是無效的 (想想從賬戶中提款的金額超過實際存在的金額的情況)。

我們可能更偏好 ev_trans' 的柯里化版本：

$$ev\_trans : Transaction \to State \to State^? \tag{4}$$

因為一個區塊本質上是交易的列表，所以區塊的評估函數

$$ev\_block : Block \to State \to State^? \tag{5}$$

可以從 ev_trans 中衍生出來。它接受一個區塊 $B : Block$ 和前一個區塊鏈狀態 $s : State$ (可能包括前一個區塊的雜湊) 並計算下一個區塊鏈狀態 $s' = ev\_block(B)(s) : State$，它要么是一個真正的狀態，要么是一個特殊值 $\perp$ 表示下一狀態無法被計算 (也就是說，從給定的起始狀態評估時該區塊是無效的——例如，該區塊包含試圖從一個空帳戶扣款的交易)。

2.2.7. Block sequence numbers. 每個在區塊鏈中的區塊 B 可以由其序列號 $\text{BLK-SEQNO}(B)$ 來參照，從第一個區塊開始為零，並在過渡到下一個區塊時加一。更正式地說，

$$\text{BLK-SEQNO}(B) = \text{BLK-SEQNO}\big(\text{BLK-PREV}(B)\big) + 1 \tag{6}$$

請注意，序列號在有分叉的情況下不能唯一識別一個區塊。

2.2.8. Block hashes. 參考區塊 B 的另一種方式是通過其雜湊 $\text{BLK-HASH}(B)$，其實際上是區塊 B 的頭部的雜湊（但是，區塊的頭部通常包含依賴於區塊 B 的所有內容的雜湊）。假設所使用的雜湊函數沒有碰撞（或至少它們是非常不可能的），一個區塊可以由其雜湊唯一識別。

2.2.9. Hash assumption. 在對區塊鏈算法進行正式分析時，我們假設使用的 k-bit 雜湊函數 $\text{HASH} : Bytes^* \to \mathbf{2}^k$ 沒有碰撞：

$$\text{HASH}(s) = \text{HASH}(s') \Rightarrow s = s' \quad \text{對任何 } s, s' \in Bytes^* \tag{7}$$

這裡的 $Bytes = \{0 \ldots 255\} = \mathbf{2}^8$ 是位元組的類型，或所有位元組值的集合，而 $Bytes^*$ 是任意（有限）位元組列表的類型或集合；而 $\mathbf{2} = \{0, 1\}$ 是位元類型，和 $\mathbf{2}^k$ 是所有 k-bit 序列的集合（即，k-bit 數字）。

當然，(7) 在數學上是不可能的，因為從一個無窮集合到一個有限集合的映射不能是單射。一個更嚴格的假設是

$$\forall s, s' : s \neq s', P\big(\text{HASH}(s) = \text{HASH}(s')\big) = 2^{-k} \tag{8}$$

然而，這對於證明不太方便。如果在某個小的 ϵ（例如，$\epsilon = 10^{-18}$）的證明中(8) 至多使用了 N 次，其中 $2^{-k}N < \epsilon$，我們可以像 (7) 是真的那樣推理，只要我們接受失敗概率 ϵ（即，最終的結論至少以概率 $1 - \epsilon$ 為真）。

最後的備註：為了使 (8) 的概率說明真正嚴格，必須在所有位元組序列的集合 $Bytes^*$ 上引入一個概率分佈。做到這一點的一種方法是假設相同長度 l 的所有位元組序列都是等概率的，並設置觀察到長度為 l 的序列的概率等於 $p^l - p^{l+1}$，對於某些 $p \to 1-$。然後應該將(8) 理解為當 p 從下面趨近於一時的條件概率 $P\big(\textsc{Hash}(s) = \textsc{Hash}(s') | s \neq s'\big)$ 的極限。

2.2.10. Hash used for the TON Blockchain. 我們目前為 TON Blockchain 使用 256-bit 的 SHA256 雜湊。如果它被證明比預期弱，未來可以被另一個雜湊函數所取代。雜湊函數的選擇是協議的可配置參數，所以可以在 **2.1.21**中解釋的，無需硬分叉即可更改。

2.3 區塊鏈狀態、帳戶和雜湊映射

如上所述，任何區塊鏈都定義了某種全局狀態，且每個區塊和每個交易都定義了這個全局狀態的轉換。在此我們描述由 TON 區塊鏈使用的全局狀態。

2.3.1. Account IDs. TON 區塊鏈使用的基本帳戶 ID或者至少由其主鏈和工作鏈零所使用 是 256-bit 整數，假設是針對特定橢圓曲線的 256-bit 橢圓曲線密碼學 (ECC) 的公鑰。這樣，

$$account\_id : Account = uint_{256} = \mathbf{2}^{256} \tag{9}$$

這裡的 $Account$ 是帳戶的類型，而 $account\_id : Account$ 是類型 $Account$ 的特定變量。

其他工作鏈可以使用其他帳戶 ID 格式，無論是 256-bit 還是其他格式。例如，可以使用等於 ECC 公鑰的 SHA256 的比特幣風格的帳戶 ID。

但是，帳戶 ID 的位長度 l 必須在工作鏈的創建期間（在主鏈中）固定，且必須至少為 64，因為 $account\_id$ 的前 64 位用於分片和訊息路由。

2.3.2. Main component: *Hashmaps*. TON 區塊鏈狀態的主要組件是一個雜湊映射。在某些情況下，我們考慮（部分定義的）“映射” $h : \mathbf{2}^n \dashrightarrow \mathbf{2}^m$。更一般地說，我們可能對於複合類型 X 的雜湊映射 $h : \mathbf{2}^n \dashrightarrow X$ 感興趣。但是，源（或索引）類型幾乎總是 $\mathbf{2}^n$。

有時，我們有一個 “預設值” $empty : X$，且雜湊映射 $h : \mathbf{2}^n \to X$ 由其 “預設值” $i \mapsto empty$「初始化」。

2.3.3. Example: TON account balances. 一個重要的例子是 TON 帳戶餘額。它是一個雜湊映射

$$balance : Account \to uint_{128} \tag{10}$$

將 $Account = \mathbf{2}^{256}$ 映射為類型為 $uint_{128} = \mathbf{2}^{128}$ 的 Gram (TON 幣) 餘額。此雜湊映射的預設值為零，這意味著最初（在處理第一個區塊之前）所有帳戶的餘額都是零。

2.3.4. Example: smart-contract persistent storage. 另一個例子是智能合約的持久存儲，可以（非常大致地）表示為一個雜湊映射

$$storage : \mathbf{2}^{256} \dashrightarrow \mathbf{2}^{256} \tag{11}$$

此雜湊映射的預設值也為零，這意味著未初始化的持久存儲單元被認為是零。

2.3.5. Example: persistent storage of all smart contracts. 因為我們有多於一個的智能合約，由 $account\_id$ 區分，每個合約都有其獨立的持久存儲，所以我們實際上必須有一個雜湊映射

$$Storage : Account \dashrightarrow (\mathbf{2}^{256} \dashrightarrow \mathbf{2}^{256}) \tag{12}$$

將智能合約的 $account\_id$ 映射到其持久存儲。

2.3.6. Hashmap type. 雜湊映射不僅僅是一個抽象的（部分定義的）函數 $\mathbf{2}^n \dashrightarrow X$；它具有特定的表示方式。因此，我們假設我們有一個特殊的雜湊映射類型

$$Hashmap(n, X) : Type \tag{13}$$

對應於編碼（部分）映射 $\mathbf{2}^n \dashrightarrow X$ 的資料結構。我們也可以寫作

$$Hashmap(n : nat)(X : Type) : Type \tag{14}$$

或

$$Hashmap : nat \to Type \to Type \tag{15}$$

我們總是可以將 $h : Hashmap(n, X)$ 轉換為一個映射 $hget(h) : \mathbf{2}^n \to X^?$。從此，我們通常寫作 $h[i]$ 而非 $hget(h)(i)$:

$$h[i] :\equiv hget(h)(i) : X^? \quad \text{對於任何 } i : \mathbf{2}^n,\ h : Hashmap(n, X) \tag{16}$$

2.3.7. Definition of hashmap type as a Patricia tree. 從邏輯上講，我們可能會定義 *Hashmap*(n, X) 為一個深度為 n 的（不完整的）二進制樹，其邊的標籤為 0 和 1，而葉子中的值類型為 X。描述相同結構的另一種方式是作為長度等於 n 的二進制字符串的 *(按位) trie*。

在實際應用中，我們更傾向於使用這種 trie 的緊湊表示，通過壓縮每個只有一個子節點的頂點及其父節點。得到的表示稱為 *Patricia tree* 或 *binary radix tree*。每個中間頂點現在都有確切的兩個子節點，由兩個非空的二進制字符串標記，左子節點開始為零，右子節點開始為一。

換句話說，在 Patricia 樹中有兩種類型的（非根）節點：

- $\textsc{Leaf}(x)$，包含類型為 X 的值 x。
- $\textsc{Node}(l, s_l, r, s_r)$，其中 l 是左子節點或子樹的（引用），s_l 是連接此頂點到其左子節點的邊的位字符串標籤（始終以 0 開頭），r 是右子樹，s_r 是到右子節點的邊的位字符串標籤（始終以 1 開頭）。

還需要第三種節點類型，只在 Patricia 樹的根上使用一次：

- $\textsc{Root}(n, s_0, t)$，其中 n 是 *Hashmap*(n, X) 的索引位字符串的公共長度，s_0 是所有索引位字符串的公共前綴，t 是指向 $\textsc{Leaf}$ 或 $\textsc{Node}$ 的引用。

如果我們想允許 Patricia 樹為空，則會使用第四種類型的（根）節點：

- $\textsc{EmptyRoot}(n)$，其中 n 是所有索引位字符串的公共長度。

我們通過以下方式定義 Patricia 樹的高度：

$$\textsc{height}(\textsc{Leaf}(x)) = 0 \tag{17}$$

$$\textsc{height}\big(\textsc{Node}(l, s_l, r, s_r)\big) = \textsc{height}(l) + \textsc{len}(s_l) = \textsc{height}(r) + \textsc{len}(s_r) \tag{18}$$

$$\textsc{height}\big(\textsc{Root}(n, s_0, t)\big) = \textsc{len}(s_0) + \textsc{height}(t) = n \tag{19}$$

最後兩個公式中的最後兩個表達式必須相等。我們使用高度為 n 的 Patricia 樹來表示類型為 *Hashmap*(n, X) 的值。

如果樹中有 N 個葉子（即，我們的雜湊映射包含 N 個值），則確切有 $N-1$ 個中間頂點。插入一個新值總是涉及通過在中間插入一個新頂點來分割一個現有的邊，並添加一個新葉子作為這個新頂點的另一個子節點。從雜湊映射中刪除一個值做的恰恰相反：葉子和它的父節點被刪除，並且父節點的父節點和其另一個子節點直接連接。

2.3.8. Merkle-Patricia trees. 當使用區塊鏈時，我們希望能夠比較 Patricia 樹（即，雜湊映射）及其子樹，並將它們縮減為單一的雜湊值。實現此目的的經典方法是由 Merkle 樹給出的。本質上，我們希望描述一種利用雜湊函數 HASH（為二進制字符串定義）對類型為 $\mathit{Hashmap}(n, X)$ 的對象 h 進行雜湊的方法，只要我們知道如何計算對象 $x : X$ 的雜湊 $\textsc{Hash}(x)$（例如，通過將雜湊函數 HASH 應用於對象 x 的二進制序列化）。

我們可能會如下遞迴地定義 $\textsc{Hash}(h)$：

$$\textsc{Hash}\big(\textsc{Leaf}(x)\big) := \textsc{Hash}(x) \tag{20}$$

$$\textsc{Hash}\big(\textsc{Node}(l, s_l, r, s_r)\big) := \textsc{Hash}\big(\textsc{Hash}(l).\,\textsc{Hash}(r).\,\textsc{code}(s_l).\,\textsc{code}(s_r)\big) \tag{21}$$

$$\textsc{Hash}\big(\textsc{Root}(n, s_0, t)\big) := \textsc{Hash}\big(\textsc{code}(n).\,\textsc{code}(s_0).\,\textsc{Hash}(t)\big) \tag{22}$$

在此，$s.t$ 表示 (位) 字符串 s 和 t 的連接，而 $\textsc{code}(s)$ 是所有位字符串 s 的前綴碼。例如，可以通過 10 來編碼 0，通過 11 來編碼 1，並通過 0 來編碼字符串的結尾。[5]

我們稍後會看到（參見 **2.3.12** 和 **2.3.14**），這是針對任意（依賴型）代數類型的值的遞迴定義的雜湊的（稍微調整的）版本。

2.3.9. Recomputing Merkle tree hashes. 這種遞迴定義 $\textsc{Hash}(h)$ 的方法，稱為 *Merkle tree hash*，具有以下優點：如果與每個節點 h' 一起明確存儲 $\textsc{Hash}(h')$（結果在結構上被稱為 *Merkle tree*，或在我們的情況下，稱為 *Merkle–Patricia tree*），則當元素被添加到雜湊映射、從雜湊映射中刪除或在雜湊映射中更改時，最多只需要重新計算 n 個雜湊。

因此，如果將全局區塊鏈狀態表示為適當的 Merkle 樹雜湊，則在每次交易後，重新計算此狀態雜湊就變得很容易。

2.3.10. Merkle proofs. 根據所選雜湊函數 HASH 的「單射性」假設 (7)，可以構造一個證明，對於 geven 值 z 的 $\textsc{Hash}(h)$, $h : \mathit{Hashmap}(n, X)$, 存在某些 $i : \mathbf{2}^n$ 和 $x : X$ 使得 $\mathit{hget}(h)(i) = x$。這樣的證明將包括從對應於 i 的葉子到根的 Merkle–Patricia 樹中的路徑，由此路徑上出現的所有節點的所有兄弟的雜湊增強。

這樣，一個輕量節點[6] 只知道某些 hashmap h 的 $\textsc{Hash}(h)$ 值（例如，智

[5]可以證明對於大約一半的 Patricia 樹的邊標籤（具有隨機或連續索引）來說，這種編碼是最優的。其餘的邊標籤可能會很長（即，幾乎有 256 位）。因此，邊標籤的幾乎最優編碼是使用上述碼，對於「短」位字符串使用前綴 0，然後編碼 1，然後是包含位字符串 s 的長度 $l = |s|$ 的九位，然後是 s 的 l 位，用於「長」位字符串（其中 $l \geq 10$）。

[6]「輕量節點」不跟踪 shardchain 的完整狀態；相反，它保留最小的資訊，例如幾個最近的區塊的雜湊，並在需要檢查完整狀態的某些部分時依賴於從完整節點獲得的資訊。

能合約的持久存儲或全局區塊鏈狀態）可能會從完整節點[7] 請求不僅僅是值 $x = h[i] = hget(h)(i)$，而是伴隨著從已知值 $\textsc{Hash}(h)$ 開始的 Merkle 證明的這樣一個值。然後，在假設 (7) 下，輕量節點可以自己檢查 x 確實是 $h[i]$ 的正確值。

在某些情況下，客戶端可能希望獲得值 $y = \textsc{Hash}(x) = \textsc{Hash}(h[i])$，例如，如果 x 本身非常大（例如，是一個 hashmap）。然後，可以提供 (i, y) 的 Merkle 證明。如果 x 也是一個 hashmap，那麼可以從完整節點獲得從 $y = \textsc{Hash}(x)$ 開始的第二個 Merkle 證明，以提供值 $x[j] = h[i][j]$ 或僅其雜湊。

2.3.11. Importance of Merkle proofs for a multi-chain system such as TON. 請注意，節點通常不能為 TON 環境中存在的所有 shardchains 成為完整節點。它通常只是某些 shardchains 的完整節點——例如，包含其自己的帳戶，它感興趣的智能合約，或者該節點已被指派為其驗證者的那些。對於其他 shardchains，它必須是一個輕量節點——否則存儲、計算和網絡帶寬的要求將是禁止的。這意味著這樣的節點不能直接檢查關於其他 shardchains 狀態的斷言；它必須依賴於從那些 shardchains 的完整節點獲得的 Merkle 證明，除非 (7) 失敗（即，找到一個雜湊碰撞），這同樣安全。

2.3.12. Peculiarities of TON VM. TON VM 或 TVM (Telegram Virtual Machine)，用於在 masterchain 和 Workchain Zero 中運行智能合約，與受到 EVM (Ethereum Virtual Machine) 啟發的常見設計有很大的不同：它不僅僅與 256 位整數工作，實際上它與（幾乎）任意的「紀錄」、「結構」或「總乘積類型」一起工作，使其更適合執行用高級（尤其是功能性）語言編寫的代碼。實際上，TVM 使用的是帶有標籤的數據類型，這與 Prolog 或 Erlang 的實現中使用的不太相同。

人們首先可以想像，TVM 智能合約的狀態不僅僅是一個 hashmap $\mathbf{2}^{256} \to \mathbf{2}^{256}$ 或 $Hashmap(256, \mathbf{2}^{256})$，但（作為第一步）是 $Hashmap(256, X)$，其中 X 是具有幾個構造器的類型，使其除了 256 位整數之外，還能存儲其他數據結構，尤其是其他的 hashmap $Hashmap(256, X)$。這意味著 TVM（持久或臨時）存儲的一個單元——或者一個在 TVM 智能合約代碼中的變量或數組元素——可能不僅包含一個整數，還包含一個全新的 hashmap。當然，這意味著一個單元不僅僅包含 256 位，還包含，例如，一個 8 位的標籤，描述如何解釋這 256 位。

事實上，值不需要確切地是 256 位的。TVM 使用的值格式由原始字節和對其他結構的引用組成，這些引用以任意順序混合，並在合適的位置插

[7]「完整節點」是跟踪有關 shardchain 的完整最新狀態的節點。

入一些描述字節，以便能夠區分指針和原始數據（例如，字符串或整數）；請參見 **2.3.14**。

這種原始值格式可以用來實現任意的總乘積代數類型。在這種情況下，該值首先包含一個原始字節，描述正在使用的「構造器」（從高級語言的角度看），然後是其他「字段」或「構造器參數」，由原始字節和對其他結構的引用組成，具體取決於選擇的構造器（參考 **2.2.5**）。然而，TVM 並不知道構造器及其參數之間的對應關係；字節和引用的混合由某些描述字節明確描述。[8]

Merkle 樹雜湊被擴展到任意這樣的結構：要計算這樣一個結構的雜湊，所有的參考都被遞歸地替換為被參考對象的雜湊，然後計算結果字節串（包括描述字節）的雜湊。

通過這種方式，對 hashmaps 的 Merkle 樹雜湊，如 **2.3.8**所述，只是用於類型 $Hashmap(n, X)$ 的兩個構造器的任意（依賴的）代數數據類型的雜湊的特殊情況。[9]

2.3.13. Persistent storage of TON smart contracts. TON 智能合約的持久性儲存主要由其「全域變數」組成，這些變數在調用智能合約之間保持不變。因此，它只是一個「產品」、「元組」或「記錄」類型，由相應於每一個全域變數的正確類型的字段組成。如果全域變數太多，由於 TON 單元大小的全局限制，它們不能放入一個 TON 單元。在這種情況下，它們被分割成幾個記錄並組織成一棵樹，基本上變成了「產品的產品」或「產品的產品的產品」類型，而不僅僅是一個產品類型。

2.3.14. TVM Cells. 最終，TON VM 在一系列的 *(TVM)* 單元中保留所有數據。每個單元首先包含兩個描述符字節，表示此單元中有多少原始數據字節（最多 128）以及有多少對其他單元的引用（最多四個）。然後是這些原始數據字節和引用。每個單元只被引用一次，所以我們可能已經在每個單元中包括了對其「父級」的引用（唯一引用此單元的單元）。但是，這個引用不必是明確的。

通過這種方式，TON 智能合約的持久數據儲存單元被組織成一棵樹，[10] 智能合約描述中保留了對這棵樹的根的引用。如果需要，可以遞歸計算這整個持久存儲的 Merkle 樹哈希，從葉子開始，然後簡單地將一個單元中的

[8]這兩個描述字節，在任何 TVM 單元中都存在，僅描述參考總數和原始字節總數；參考總是放在所有原始字節之前或之後。

[9]實際上，Leaf 和 Node 是輔助類型 $HashmapAux(n, X)$ 的構造器。類型 $Hashmap(n, X)$ 有構造器 Root 和 EmptyRoot，其中 Root 包含類型 $HashmapAux(n, X)$ 的值。

[10]邏輯上；在 **2.5.5**中描述的「單元包」表示法識別所有重複的單元，當序列化時，將此樹轉換為一個有向無環圖（dag）。

所有引用替換為所引用的單元的遞歸計算的哈希，然後計算所得字節串的哈希。

2.3.15. Generalized Merkle proofs for values of arbitrary algebraic types. 由於 TON VM 通過由 (TVM) 單元組成的樹來表示任意代數類型的值，且每個單元都有一個明確定義的（遞歸計算的）Merkle 哈希，實際上依賴於此單元為根的整個子樹，我們可以為任意代數類型的值（的部分）提供「一般化的 Merkle 證明」，旨在證明具有已知 Merkle 哈希的樹的某個子樹具有特定值或具有特定哈希的值。這概括了 **2.3.10** 的方法，其中只考慮了 $x[i] = y$ 的 Merkle 證明。

2.3.16. Support for sharding in TON VM data structures. 我們剛剛概述了如何在不過於複雜的情況下，TON VM 支持高級智能合約語言中的任意（依賴）代數數據類型。然而，對於大型（或全局）智能合約的分片需要在 TON VM 級別上的特殊支援。為此，系統中增加了 hashmap 類型的特殊版本，相當於一個「映射」$Account \dashrightarrow X$。這個「映射」可能看起來等同於 $Hashmap(m, X)$，其中 $Account = \mathbf{2}^m$。但是，當一個分片被分成兩個，或兩個分片被合併時，這樣的 hashmaps 會自動被分成兩個，或合併回來，以保留只屬於相應分片的鍵。

2.3.17. Payment for persistent storage. TON 區塊鏈的一個值得注意的特點是從智能合約中扣除用於存儲其持久數據的支付（即，增加區塊鏈的總狀態）。它的工作原理如下：

每個區塊都宣布兩種費率，以區塊鏈的主要貨幣（通常是 Gram）來提名：保持一個單元在持久儲存中的價格，以及在持久儲存的某個單元中保持一個原始字節的價格。每個賬戶使用的單元和字節的總數據作為其狀態的一部分存儲，所以通過將這些數字乘以在區塊頭中宣布的兩個費率，我們可以計算從賬戶餘額中扣除的支付，以保持其數據在前一個區塊和當前區塊之間。

然而，對於每個賬戶和每個智能合約在每個區塊中的持久儲存使用的支付並不是每次都收取的；而是在賬戶數據中存儲上次收取此支付的區塊的序列號，並且當對賬戶進行任何操作時（例如，轉移價值或接收並由智能合約處理一條消息），在執行任何進一步的操作之前，從賬戶餘額中扣除自上次這樣的支付以來的所有區塊的儲存使用支付。如果賬戶的餘額在此之後變為負數，則該賬戶將被銷毀。

一個工作鏈可能宣稱每個賬戶的一些原始數據字節是「免費的」(即，不參與持久儲存支付)，以使「簡單」的賬戶，只在一兩種加密貨幣中保持它們的餘額，免於這些常數支付。

請注意，如果沒有人給一個賬戶發送任何消息，它的持久儲存支付不會被收集，並且它可以無限期地存在。然而，任何人都可以發送，例如，一條空消息來銷毀這樣的賬戶。可以給發送這樣一條消息的人提供一個小的激勵，從要被銷毀的賬戶的原始餘額中收取部分資金。然而，我們預期，驗證者會免費銷毀這樣的無資金賬戶，僅僅是為了減少全球區塊鏈的狀態大小，並避免保持大量的數據而不得到賠償。

為持久數據的保持收集的支付在 shardchain 或 masterchain 的驗證者之間分配（在後一種情況下按比例分配他們的股份）。

2.3.18. Local and global smart contracts; smart-contract instances. 一個智能合約通常只存在於一個分片中，根據智能合約的 $account\_id$ 選擇，與「普通」賬戶類似。這通常對大多數應用程序來說都是足夠的。然而，一些「高負載」的智能合約可能希望在某個工作鏈的每個分片鏈中都有一個「實例」。為了實現這一點，它們必須將它們的創建交易傳播到所有的分片鏈中，例如，通過將此交易提交到工作鏈 w 的「根」分片鏈 $(w, \emptyset)$ 中，並支付一大筆佣金。[11]

這個動作實際上在每個分片中創建了智能合約的實例，具有單獨的餘額。原始地，創建交易中傳輸的餘額只是通過給分片 (w, s) 的實例 $2^{-|s|}$ 的總餘額的部分來分配。當一個分片分裂成兩個子分片時，所有全球智能合約的實例的餘額都分裂為一半；當兩個分片合併時，餘額加在一起。

在某些情況下，分裂/合併全球智能合約的實例可能涉及這些智能合約的特殊方法的（延遲）執行。默認情況下，餘額按照上述方式分裂和合併，一些特殊的「賬戶索引」的 hashmaps 也是自動分裂和合併的（參見 **2.3.16**）。

2.3.19. Limiting splitting of smart contracts. 一個全球智能合約可以在其創建時限制其分裂深度 d，以使持久存儲費用更具可預測性。這意味著，如果分片鏈 (w, s) 滿足 $|s| \geq d$ 被分裂成兩個，只有兩個新分片鏈中的一個繼承智能合約的實例。這個分片鏈是確定性選擇的：每個全球智能合約都有一個「$account\_id$」，本質上是其創建交易的哈希，並且其實例具有與前 $\leq d$ 位替換為適當值的相同的 $account\_id$，以落入正確的分片。這個 $account\_id$ 選擇了分裂後哪個分片將繼承智能合約實例。

2.3.20. Account/Smart-contract state. 我們可以總結以上所有內容，得出賬戶或智能合約的狀態包括以下內容：

- 區塊鏈的主要貨幣的餘額
- 區塊鏈其他貨幣的餘額

[11] 一個更昂貴的選擇是在主鏈中發布這樣一個「全球」智能合約。

- 智能合約的代碼（或其哈希）
- 智能合約的持久性數據（或其 Merkle 哈希）
- 持久性存儲單元和原始字節使用數量的統計
- 上次收取智能合約持久性存儲付款的時間（實際上，是主鏈區塊號）
- 轉移貨幣和從此賬戶發送消息所需的公鑰（可選;默認等於 *account_id* 本身）。在某些情況下，更為複雜的簽名檢查代碼可能位於此處，與比特幣交易輸出類似；然後，*account_id* 將等於此代碼的哈希。

我們還需要在某處保存以下數據，無論是在賬戶狀態中還是在某個其他的賬戶索引的哈希圖中：

- 賬戶的輸出消息隊列（參見 **2.4.17**）
- 最近傳送消息的（哈希的）集合（參見 **2.4.23**）

並不是每個賬戶都真正需要所有這些；例如，只有智能合約需要智能合約代碼，而「簡單」的賬戶則不需要。此外，儘管任何賬戶必須在主要貨幣中有一個非零餘額（例如，基礎工作鏈的主鏈和分片鏈的 Grams），但在其他貨幣中可能有零餘額。為了避免保留未使用的數據，定義了一個乘積型別（取決於工作鏈）（在工作鏈的創建期間），它使用不同的標籤字節（例如，TL 構造器；參見 **2.2.5**）來區分使用的不同「構造器」。最終，賬戶狀態本身作為 TVM 持久性存儲的單元集合保存。

2.4 分片鏈間的消息

TON 區塊鏈的一個重要組件是區塊鏈間的消息系統。這些區塊鏈可以是同一工作鏈的分片鏈，或者是不同工作鏈的分片鏈。

2.4.1. Messages, accounts and transactions: a bird's eye view of the system. 消息從一個賬戶發送到另一個賬戶。每一個交易包括一個賬戶接收一個消息，根據某些規則改變其狀態，並生成到其他賬戶的多個（可能是一個或零個）新消息。每條消息生成並接收（傳遞）確切一次。

這意味著消息在系統中扮演了基本的角色，與賬戶（智能合約）的角色相當。從無窮分片範式的角度看（參見 **2.1.2**），每個賬戶都位於其獨立的「賬戶鏈」中，並且它唯一可以影響某其他賬戶的狀態的方式是通過發送消息。

2.4.2. Accounts as processes or actors; Actor model. 可以將賬戶（和智能合約）視為「進程」或「角色」，它們能夠處理進入的消息、改變其內部狀態，並因此生成一些出站消息。這與所謂的 *Actor model* 密切相關，該模型在 Erlang 之類的語言中使用（但是，Erlang 中的角色通常稱為「進程」）。由於新的角色（即，智能合約）也允許由現有角色作為處理入站消息的結果來創建，因此與 Actor model 的對應基本上是完整的。

2.4.3. Message recipient. 任何消息都有其接收者，由目標工作鏈識別符 w（默認情況下與原始分片鏈相同）和接收賬戶 $account\_id$ 進行描述。$account\_id$ 的確切格式（即，位數）取決於 w；但是，碎片始終由其第一個（最重要的）64 位確定。

2.4.4. Message sender. 在大多數情況下，消息都有一個發送者，再次由 $(w', account\_id')$ 對進行描述。如果存在，它位於消息接收者和消息值之後。有時，發送者不重要，或者他是區塊鏈之外的某人（即，不是智能合約），在這種情況下，此字段不存在。

值得注意的是，Actor model 並不要求消息有一個隱式發送者。相反，消息可能包含對應該發送請求回覆的 Actor 的引用；它通常與發送者相符。但是，在加密貨幣（Byzantine）環境中，在消息中有一個明確的不可偽造的發送者字段是很有用的。

2.4.5. Message value. 消息的另一個重要特性是其附加的值，它是源工作鏈和目標工作鏈均支持的一種或多種加密貨幣。消息的值在消息接收者之後的開頭處指示；它實際上是一系列的 $(currency\_id, value)$ 對。

請注意，「簡單」賬戶之間的「簡單」值轉移只是帶有某些值的空（無操作）消息。另一方面，略為複雜的消息主體可能包含一個簡單的文本或二進制評論（例如，關於付款的目的）。

2.4.6. External messages, or "messages from nowhere". 有些消息是從「無處」來的，也就是它們並非由位於區塊鏈中的賬戶（無論是否是智能合約）生成的。最典型的例子是當用戶希望從她控制的賬戶轉移一些資金到另一個賬戶時。在這種情況下，用戶發送一個「來自無處的消息」到她自己的賬戶，要求它生成一個發送給接收賬戶的消息，帶有指定的值。如果此消息被正確簽名，她的賬戶就會接收到它並生成所需的出站消息。

實際上，人們可能會認為「簡單」賬戶是帶有預定義代碼的智能合約的特例。這種智能合約只接收一種類型的消息。這種入站消息必須包含作為傳遞（處理）入站消息結果要生成的出站消息列表，以及一個簽名。智能合約檢查簽名，如果它是正確的，則生成所需的消息。

當然，「來自無處的消息」和普通消息之間有所不同，因為「來自無處的消息」不能承載值，所以它們不能為自己的「gas」（即它們的處理）付款。

相反，它們在甚至被建議包含在新的 shardchain 塊中之前，會先嘗試執行一個小的 gas 限制；如果執行失敗（簽名不正確），則「來自無處的消息」被視為不正確並被丟棄。如果執行在小的 gas 限制內不失敗，則消息可能被包含在新的 shardchain 塊中並完全處理，從接收者的賬戶中扣除所消耗的 gas（處理能力）的付款。「來自無處的消息」也可以定義一些交易費，這些費用在 gas 付款之外從接收者的賬戶中扣除，以重新分配給驗證者。

在這個意義上，「來自無處的消息」或「外部消息」起到了在其他區塊鏈系統中使用的交易候選人的作用（例如，比特幣和以太坊）。

2.4.7. Log messages, or "messages to nowhere". 同樣，有時可以生成並路由到特定的 shardchain 的特殊消息，不是為了交付給其收件人，而是為了記錄，以便任何人接收到有關該碎片的更新時都可以輕鬆觀察。這些記錄的消息可以在用戶的控制台中輸出，或觸發某個離鏈伺服器上的某個腳本的執行。在這種意義上，它們代表了「區塊鏈超級電腦」的外部「輸出」，正如「來自無處的消息」代表了「區塊鏈超級電腦」的外部「輸入」。

2.4.8. Interaction with off-chain services and external blockchains. 這些外部輸入和輸出消息可以用於與離鏈服務和其他（外部）區塊鏈互動，如比特幣或以太坊。人們可以在 TON 區塊鏈中創建與比特幣、以太或在以太坊區塊鏈中定義的任何 ERC-20 代幣相關聯的代幣或加密貨幣，並使用「來自無處的消息」和「去無處的消息」，由位於某些第三方離鏈伺服器上的腳本生成和處理，以實現 TON 區塊鏈與這些外部區塊鏈之間的必要互動。

2.4.9. Message body. *message body* 基本上就是一系列的位元組，其含義只由接收的工作鏈和/或智能合約決定。對於使用 TON VM 的區塊鏈，這可以是通過 `Send()` 操作自動生成的任何 TVM cell 的序列化。這種序列化是通過遞迴地替換 TON VM cell 中的所有參照來獲得的。最終，會出現一串原始位元組，通常在前面加上一個 4 位元組的「消息類型」或「消息建構器」，用於選擇接收智能合約的正確方法。

另一種選擇是使用 TL-序列化對象（參見 **2.2.5**）作為消息主體。這對於不同工作鏈之間的通信可能尤其有用，其中一個或兩個不一定使用 TON VM。

2.4.10. Gas limit and other workchain/VM-specific parameters. 有時消息需要攜帶有關 gas 限制、gas 價格、交易費用和相似值的資訊，這些值取決於接收的工作鏈，並且只與接收的工作鏈有關，但不一定與原始工作鏈有關。這些參數包含在消息主體中或之前，有時（取決於工作鏈）有特殊的 4 位元組前綴表示它們的存在（可以由 TL-方案定義；參見 **2.2.5**）。

2.4.11. Creating messages: smart contracts and transactions. 新消息的來源有兩個。大多數消息是在智能合約執行期間創建的（通過 TON VM 中的 `Send()` 操作），當某個智能合約被調用以處理一個入站消息時。或者，消息可能來自外部，作為「外部消息」或「來自無處的消息」（參見 **2.4.6**）。[12]

2.4.12. Delivering messages. 當一條消息到達包含其目的賬戶的 shardchain 時，[13] 它被「傳遞」到其目的地賬戶。接下來會發生什麼取決於工作鏈；從外部觀點看，重要的是這樣的消息從這個 shardchain 永遠不會被進一步轉發。

對於基礎工作鏈的 shardchains，傳送包括將消息值（扣除任何 gas 付款）添加到接收賬戶的餘額，並可能之後調用接收智能合約的一個依賴於消息的方法，如果接收賬戶是一個智能合約。事實上，一個智能合約只有一個進入點用於處理所有傳入消息，並且它必須通過查看它們的前幾個位元組來區分不同類型的消息（例如，包含 TL 建構器的前四個位元組；參見 **2.2.5**）。

2.4.13. Delivery of a message is a transaction. 因為消息的交付更改了賬戶或智能合約的狀態，所以它在接收的 shardchain 中是一個特殊的交易，並明確地註冊為這樣。本質上，所有 TON 區塊鏈交易都包括將一個入站消息傳遞給其接收賬戶（智能合約），忽略一些次要技術細節。

2.4.14. Messages between instances of the same smart contract. 回憶一下，一個智能合約可能是本地 的（即，像任何普通賬戶一樣駐留在一個 shardchain 中）或全局 的（即，在所有的 shards 中都有實例，或至少在所有深度為 d 的 shards 中；參見 **2.3.18**）。全球智能合約的實例可能需要交換特殊消息以在彼此之間傳遞信息和價值。在這種情況下，（不可偽造的）發件人 *account_id* 變得很重要（參見 **2.4.4**）。

2.4.15. Messages to any instance of a smart contract; wildcard addresses. 有時候一條消息（例如，客戶端請求）需要被交付給全球智能合約的任何實例，通常是最近的一個（如果有一個駐留在與發件人相同的 shardchain 中，它是明顯的候選人）。做到這一點的一種方法是使用「通配符收件人地址」，其中目標 *account_id* 的前 d 位可以取任意值。實際上，人們通常會將這 d 位設置為與發件人的 *account_id* 中的相同值。

[12] 上述只需要在基本工作鏈及其分片鏈上文字上是真實的；其他工作鏈可能提供其他創建消息的方法。

[13] 作為一個退化情況，這個 shardchain 可能與原始的 shardchain 重合——例如，如果我們正在內部工作的工作鏈尚未被分裂。

2.4.16. Input queue is absent. 由區塊鏈接收的所有消息（通常是 shardchain；有時是 masterchain）——或基本上是駐留在某個 shardchain 內的「賬戶鏈」——都立即被交付（即，由接收賬戶處理）。因此，沒有像這樣的「輸入隊列」。相反，如果由於對區塊總大小和 gas 使用的限制，不是所有目的為特定 shardchain 的消息都可以被處理，一些消息簡單地留在原始 shardchains 的輸出隊列中積累。

2.4.17. Output queues. 從無限分割範例（Infinite Sharding Paradigm）的角度看 (cf. **2.1.2**)，每個賬戶鏈（即，每個賬戶）都有其自己的輸出隊列，由它生成但尚未傳遞給其接收者的所有消息組成。當然，賬戶鏈只有一個虛擬的存在；它們被分組到 shardchains 中，而一個 shardchain 有一個輸出「隊列」，由屬於該 shardchain 的所有賬戶的輸出隊列的聯合組成。

這個 shardchain 輸出「隊列」只對其成員消息施加部分順序。即，在先前的區塊中生成的消息必須在隨後的區塊中生成的任何消息之前被傳遞，並且由相同賬戶生成且具有相同目的地的任何消息必須按照它們的生成順序傳遞。

2.4.18. Reliable and fast inter-chain messaging. 對於像 TON 這樣的可擴展多區塊鏈項目，能夠在不同的 shardchains 之間轉發和傳遞消息是至關重要的 (cf. **2.1.3**)，即使系統中有數百萬個。消息應該被可靠地（即，消息不應該丟失或傳遞多次）和快速地傳遞。TON 區塊鏈通過使用兩種「消息路由」機制的組合來實現這一目標。

2.4.19. Hypercube routing: "slow path" for messages with assured delivery. TON Blockchain 使用「超立方體路由」作為一種緩慢，但安全可靠的方法，從一個 shardchain 傳遞消息到另一個 shardchain，如有必要，使用幾個中間的 shardchain 進行轉發。否則，任何給定的 shardchain 的驗證者都需要追踪所有其他 shardchain 的狀態（輸出隊列），隨著 shardchain 總數的增加，這將需要過多的計算能力和網絡帶寬，從而限制了系統的可擴展性。因此，無法直接從任何 shard 傳遞消息到其他每一個 shard。相反，每個 shard 只與其 (w, s) shard 標識符中確切一個十六進制數位不同的 shard「連接」(cf. **2.1.8**)。這樣，所有的 shardchain 構成一個「超立方體」圖，消息沿著這個超立方體的邊緣移動。

如果消息被發送到與當前的 shard 不同的 shard，當前的 shard 標識符的十六進制數字（確定地選擇）被替換為目標 shard 的相應數字，並使用所得的標識符作為轉發消息的近端目標。[14]

[14]這不一定是用於計算超立方體路由的下一跳的算法的最終版本。特別是，十六進制數字可能會被 r-位群組替換，其中 r 是一個可配置的參數，不一定等於四。

超立方體路由的主要優點是區塊有效性條件意味著創建 shardchain 的區塊的驗證者必須收集和處理「鄰近」shardchain 的輸出隊列中的消息，否則將失去他們的賭注。這樣，任何消息都可以預期最終會達到其最終目的地；消息不能在過程中丟失或傳遞兩次。

請注意，超立方體路由帶來了一些額外的延遲和開銷，因為必須通過幾個中間的 shardchain 轉發消息。然而，這些中間的 shardchain 的數量增長非常緩慢，作為 shardchain 總數 N 的對數 $\log N$（更確切地說，$\lceil \log_{16} N \rceil - 1$）。例如，如果 $N \approx 250$，最多只有一個中間的 hop；對於 $N \approx 4000$ 的 shardchain，最多有兩個。四個中間的 hop，我們可以支持多達一百萬的 shardchain。我們認為為系統的基本無限的可擴展性支付的這個代價是非常小的。事實上，甚至不必支付這個價格：

2.4.20. Instant Hypercube Routing: "fast path" for messages. TON Blockchain 的一個新功能是它引入了一個「快速路徑」，從一個 shardchain 轉發消息到任何其他 shardchain，在大多數情況下都可以完全繞過 **2.4.19** 中的「慢速」超立方體路由，並在最終目的地 shardchain 的下一個區塊中傳遞消息。

該思路如下。在「慢速」的超立方體路由中，消息在超立方體的邊緣上（在網絡中）旅行，但在每個中間頂點都會被延遲（大約五秒鐘），以將其提交到相應的 shardchain，然後再繼續其旅程。

為了避免不必要的延遲，可以沿著超立方體的邊緣轉發消息和一個適當的 Merkle 證明，而不用等待將其提交到中間的 shardchains。實際上，網絡消息應該從原始 shard 的「任務組」的驗證者（cf. **2.6.8**）轉發到目的地 shard 的「任務組」的指定區塊生產者（cf. **2.6.9**）；這可能可以直接完成，而不沿著超立方體的邊緣。當這條帶有 Merkle 證明的消息到達目的地 shardchain 的驗證者（更確切地說，是 collators; cf. **2.6.5**）時，他們可以立即將其提交到一個新的區塊，而不用等待消息完成沿著「慢路徑」的旅程。然後將交付確認以及一個適當的 Merkle 證明沿著超立方體的邊緣發回，並且可以通過提交一個特殊的交易來停止消息沿著「慢路徑」的旅程。

請注意，這種「即時交付」機制並未取代在 **2.4.19** 中描述的「慢速」但是不會失敗的機制。仍然需要「慢路徑」，因為驗證者不能因失去或簡單地決定不將「快速路徑」消息提交到他們區塊鏈的新區塊而受到懲罰。[15]

因此，兩種消息轉發方法是平行運行的，只有在「快速」機制的成功證明被提交到一個中間的 shardchain 時，「慢速」機制才會被中止。

2.4.21. Collecting input messages from output queues of neighbor-

[15]然而，驗證者有一定的動機盡快這樣做，因為他們將能夠收集與消息相關的所有轉發費用，這些費用尚未在慢路徑中被消耗。

ing shardchains. 當為 shardchain 提議一個新的區塊時，鄰近（在**2.4.19**的路由超立方體意義上）的 shardchains 的一些輸出消息被包含在新的區塊中作為「輸入」消息，並立即被傳遞（即，處理）。關於這些鄰居的輸出消息必須以哪種順序進行處理，有一定的規則。基本上，一個「較舊」的消息（來自參照較舊的主鏈區塊的 shardchain 區塊）必須在任何「較新」的消息之前傳遞；對於來自同一鄰近 shardchain 的消息，必須遵守**2.4.17**中描述的輸出隊列的部分順序。

2.4.22. Deleting messages from output queues. 一旦觀察到輸出隊列中的訊息已被相鄰的分片鏈交付，則通過特殊交易明確地從輸出隊列中刪除它。

2.4.23. Preventing double delivery of messages. 為了防止從相鄰的分片鏈的輸出隊列中雙重交付訊息，每個分片鏈（更確切地說，其中的每個賬戶鏈）作為其狀態的一部分保持最近交付的訊息的集合（或僅其哈希值）。當觀察到已交付的訊息從其源相鄰分片鏈（參見**2.4.22**）的輸出隊列中被刪除時，它也從最近交付的訊息的集合中被刪除。

2.4.24. Forwarding messages intended for other shardchains. 通過 Hypercube 路由（參見**2.4.19**）有時出站訊息不是交付給包含預期收件人的分片鏈，而是交付給位於到目的地的超立方體路徑上的相鄰分片鏈。在這種情況下，" 交付" 包括將入站訊息移動到出站隊列。這在塊中明確地反映為一個特殊的轉發交易，其中包含該訊息。本質上，這看起來就像該訊息已被分片鏈內的某人接收，並且生成了一個相同的訊息作為結果。

2.4.25. Payment for forwarding and keeping a message. 轉發交易實際上消耗了一些瓦斯（取決於被轉發的訊息的大小），因此從被轉發的訊息的值中扣除了一筆瓦斯支付，代表此分片鏈的驗證器。此轉發支付通常遠小於當訊息最終交付給其收件人時所確定的瓦斯支付，即使該訊息由於超立方體路由而被轉發了多次。此外，只要某個分片鏈的輸出隊列中保持有訊息，它就是分片鏈的全局狀態的一部分，因此特殊交易也可能收取長時間保持全局數據的支付。

2.4.26. Messages to and from the masterchain. 訊息可以直接從任何分片鏈發送到主鏈，反之亦然。但是，發送訊息到主鏈以及在主鏈中處理訊息的瓦斯價格相當高，因此只有在真正需要時才會使用此功能——例如，由驗證器來存入他們的賭注。在某些情況下，可能會定義發送到主鏈的訊息的最小存款（附加值），只有當接收方認為該訊息是“有效”的時候才會退還。

訊息不能自動通過主鏈路由。帶有 $workchain\_id \neq -1$ 的訊息（其中 -1 是表示主鏈的特殊 $workchain\_id$）不能交付給主鏈。

原則上，人們可以在主鏈內部創建一個訊息轉發智能合約，但使用它的價格將是禁止性的。

2.4.27. Messages between accounts in the same shardchain. 在某些情況下，某個分片鏈中的賬戶生成的訊息，目標是同一分片鏈中的另一賬戶。例如，這發生在新的工作鏈中，因為負載是可管理的，所以尚未分裂成多個分片鏈。

這樣的訊息可能會在分片鏈的輸出隊列中累積，然後在後續的區塊中作為入站訊息進行處理（為此目的，任何分片都被視為其本身的鄰居）。然而，在大多數情況下，有可能在起源區塊本身內交付這些訊息。

為了實現這一點，對包含在分片鏈區塊中的所有交易施加了部分排序，並尊重此部分順序處理交易（每個交易都包含將訊息交付給某個賬戶）。特別是，允許一個交易處理與此部分順序相對的前一個交易的某個輸出訊息。

在這種情況下，訊息主體不會被複製兩次。相反，起源交易和處理交易都參照訊息的共享副本。

2.5 Global Shardchain State. "Bag of Cells" Philosophy.

現在我們準備描述 TON 區塊鏈的全局狀態，或者至少是基本工作鏈的分片鏈。

我們從"高級"或"邏輯"描述開始,即全局狀態是代數類型 *ShardchainState* 的值。

2.5.1. Shardchain state as a collection of account-chain states. 根據無限分片模型（參見**2.1.2**），任何分片鏈只是虛擬"賬戶鏈"的（臨時）集合，每個賬戶鏈只包含一個賬戶。這意味著，本質上，全局分片鏈狀態必須是一個哈希映射

$$ShardchainState := (Account \dashrightarrow AccountState) \tag{23}$$

其中所有作為此哈希映射的指數出現的 $account\_id$ 必須以前綴 s 開始，如果我們正在討論分片 (w, s) 的狀態（參見**2.1.8**）。

實際上，我們可能希望將 *AccountState* 分成幾個部分（例如，保持賬戶輸出訊息隊列的獨立，以簡化相鄰分片鏈的檢查），並在 *ShardchainState* 內部擁有幾個哈希映射 $(Account \dashrightarrow AccountStatePart_i)$。我們還可能向

ShardchainState 添加少量“全局”或“整體”參數，（例如，屬於此分片的所有賬戶的總餘額，或所有輸出隊列中的訊息總數）。

然而，(23) 是分片鏈全局狀態的一個很好的初步近似值，至少從“邏輯”（“高級”）的角度來看。可以使用 TL 方案（參見**2.2.5**）的幫助來進行代數類型 *AccountState* 和 *ShardchainState* 的正式描述，該描述將在其他地方提供。

2.5.2. Splitting and merging shardchain states. 請注意，無限分片模型描述的分片鏈狀態 (23) 顯示了當分片被分裂或合併時，該狀態應如何被處理。實際上，這些狀態變換變得是非常簡單的哈希映射操作。

2.5.3. Account-chain state. (虛擬的) 賬戶鏈狀態只是一個賬戶的狀態，由類型 *AccountState* 描述。通常它具有在 **2.3.20** 中列出的所有或某些字段，具體取決於使用的構造器。

2.5.4. Global workchain state. 與 (23) 類似，我們可以使用相同的公式定義全局的 *workchain* 狀態，但 *account_id*'s 可以取任何值，不僅僅是屬於一個分片的值。在這種情況下，也適用於 **2.5.1** 中所做的類似備註：我們可能想要將此哈希映射分裂成幾個哈希映射，我們可能想要添加一些“整體”的參數，如總餘額。

本質上，全局的工作鏈狀態 *must* 被給予與分片鏈狀態相同的類型 *ShardchainState*，因為如果這個工作鏈的所有現有分片鏈突然合併成一個，我們會得到的就是這個分片鏈狀態。

2.5.5. Low-level perspective: "bag of cells". 存在一個「低階」描述關於賬戶鏈或 shardchain 狀態，這與上述的「高階」描述是互補的。此描述相當重要，因為它事實上是非常普遍的，為代表、儲存、序列化和通過網路轉移 TON Blockchain（包括 blocks、shardchain states、smart-contract storage、Merkle proofs 等）所用的幾乎所有資料提供了一個共同的基礎。同時，一旦這樣的普遍「低階」描述被理解和實施，我們就可以集中注意力僅考慮「高階」。

回想一下，TVM 用一棵「TVM cells」樹，或簡稱為「cells」（參見 **2.3.14**和 **2.2.5**）來表示任意代數類型的值（例如， (23)中的 *ShardchainState*）。

每個這樣的 cell 都由兩個「descriptor bytes」組成，定義某些標誌和值 $0 \le b \le 128$，表示原始 bytes 的數量，以及 $0 \le c \le 4$，這是指向其他 cells 的引用數量。然後是 b 個原始 bytes 和 c 個 cell 引用。[16]

[16]可以顯示，如果經常需要儲存在 cell 樹中的所有資料的 Merkle proofs，則應該使用 $b + ch \approx 2(h + r)$ 的 cells 來最小化平均 Merkle proof 大小，其中 $h = 32$ 是 hash 在 bytes 中的大小，而 $r \approx 4$ 是 cell 引用的「byte 大小」。換句話說，一個 cell 應該包含兩個引用和一些原始 bytes，或一個引用和大約 36 原始 bytes，或完全沒有引用但是有 72 原始 bytes。

cell 引用的確切格式取決於它的實現，以及 cell 是否位於 RAM、磁碟、網絡封包、block 等。一個有用的抽象模型是想像所有 cells 都存放在內容可寄存的記憶體中，cell 的地址等於其 (SHA256) hash。回想一下，cell 的 (Merkle) hash 正是通過將其子 cell 的引用替換為它們（遞迴計算的）hashes 並 hash 生成的 byte string 來計算的。

這樣，如果我們使用 cell hashes 來引用 cells（例如，在其他 cells 的描述中），系統稍微簡化，且 cell 的 hash 開始與代表它的 byte string 的 hash 相符。

現在我們看到，任何 *TVM* 可以表示的對象，包括全局 *shardchain* 狀態，都可以表示為一個「*bag of cells*」——即，一個 *cells* 的集合以及指向其中之一的「*root*」引用（例如，通過 hash）。請注意，重複的 cells 從此描述中被刪除了（「bag of cells」是 cells 的集合，而不是多重集），所以抽象的樹表示實際上可能變成了一個有向無環圖 (dag) 表示。

人甚至可以在硬碟上使用 *B*-tree 或 *B*+-tree 來保存這個狀態，包含所有相關的 cells（也許還有一些附加數據，如子樹高度或引用計數器），並由 cell hash 進行索引。但是，這個想法的簡單實現會導致一個智能合約的狀態被散佈在磁盤文件的遠處，這是我們想要避免的。[17]

現在我們將詳細解釋 TON Blockchain 使用的幾乎所有對象如何可以表示為「bag of cells」，從而展示這種方法的普遍性。

2.5.6. Shardchain block as a "bag of cells". Shardchain block 本身也可以用代數類型來描述，並存儲為「bag of cells」。然後可以通過簡單地連接表示「bag of cells」中每個 cell 的 byte strings（以任意順序）來獲得 block 的簡單二進制表示。這種表示可以進一步改進和優化，例如，在 block 的開頭提供所有 cells 的偏移量列表，並在可能的情況下用 32 位索引替換對其他 cells 的 hash 引用。然而，人們應該認識到 block 本質上是一個「bag of cells」，所有其他技術細節只是次要的優化和實現問題。

2.5.7. Update to an object as a "bag of cells". 想像我們有一個以「bag of cells」表示的對象的舊版本，我們想要表示同一對象的新版本，這個新版本應該與前一個不太不同。一個方法是簡單地將新狀態表示為具有自己 root 的另一個「bag of cells」，並從中刪除所有在舊版本中出現的 *cells*。剩下的「bag of cells」基本上是對象的 *update*。每個擁有此對象的舊版本和 update 的人都可以計算新版本，只需合併兩個 bag of cells，並刪除舊的 root（減少其引用計數，並在引用計數變為零時釋放 cell）。

[17]更好的實現方法是，如果 smart contract 的狀態很小，就將其保存為序列化的字符串，如果很大，則保存在另一個 *B*-tree 中；然後代表 blockchain 狀態的頂層結構將是一個 *B*-tree，其葉子節點被允許包含對其他 *B*-tree 的引用。

2.5.8. Updates to the state of an account. 特別是，對賬戶的狀態、shardchain 的全局狀態或任何 hashmap 的更新都可以使用在 **2.5.7**中描述的想法進行表示。這意味著當我們接收到一個新的 shardchain block（即是「bag of cells」）時，我們不只是單獨解釋這個「bag of cells」，而是首先將其與代表 shardchain 先前狀態的「bag of cells」結合。從這個意義上說，每個 block 可能都「包含」blockchain 的整體狀態。

2.5.9. Updates to a block. 回憶一下，block 本身就是一個「bag of cells」，所以，如果需要編輯 block，則可以類似地將「block update」定義為「bag of cells」，並在存在該 block 的先前版本的「bag of cells」的情境下進行解釋。這大致上是在 **2.1.17**中討論的「垂直 blocks」背後的想法。

2.5.10. Merkle proof as a "bag of cells". 注意，一個（廣義的）Merkle 證明——例如，從已知的 $\textsc{Hash}(x) = h$ 開始聲稱 $x[i] = y$（參見 **2.3.10**和 **2.3.15**）——也可以表示為「bag of cells」。具體來說，只需要提供一組 cells 子集，對應從 $x : \mathit{Hashmap}(n, X)$ 的根到其所需的具有索引 $i : \mathbf{2}^n$ 和值 $y : X$ 的葉子的路徑。在此證明中，不位於此路徑上的這些 cells 的子項的引用將保持「未解決」，由 cell hashes 表示。還可以同時提供，例如，$x[i] = y$ 和 $x[i'] = y'$ 的 Merkle 證明，通過在「bag of cells」中包括位於從 x 的根到對應於索引 i 和 i' 的葉子的兩條路徑的聯合上的 cells。

2.5.11. Merkle proofs as query responses from full nodes. 實質上，擁有 shardchain（或 account-chain）狀態完整副本的完整節點可以在被輕節點（例如，運行 TON Blockchain 客戶端輕版本的網絡節點）請求時提供 Merkle 證明，使接收者能夠僅使用此 Merkle 證明中提供的 cells 執行一些簡單的查詢，而不需要外部幫助。輕節點可以將其查詢以序列化格式發送給完整節點，並接收正確的答案和 Merkle 證明——或僅僅是 Merkle 證明，因為請求者應該能夠僅使用 Merkle 證明中包含的 cells 來計算答案。這個 Merkle 證明將僅由一個「bag of cells」組成，只包含屬於 shardchain 狀態的那些 cells，在執行輕節點的查詢時由完整節點訪問。此方法尤其可用於執行智能合約的「get queries」（參見 **5.3.12**）。

2.5.12. Augmented update, or state update with Merkle proof of validity. 回想一下（參見 **2.5.7**），我們可以透過「update」來描述從舊值 $x : X$ 到新值 $x' : X$ 的物件狀態變化，這只是一個「bag of cells」，其中包含那些位於表示新值 x' 的子樹中的 cells，但不包含位於表示舊值 x 的子樹中的 cells，因為假設接收者擁有舊值 x 及其所有的 cells 複本。

然而，如果接收者並沒有 x 的完整複本，而只知道其（Merkle）hash $h = \textsc{Hash}(x)$，它將無法檢查 update 的有效性（即 update 中的所有「懸

空」cell 引用確實指向存在於 x 的樹中的 cells)。我們希望能夠具有可驗證性的 update，並加上對舊狀態中所有引用 cells 存在的 Merkle 證明。這樣，只知道 $h = \text{Hash}(x)$ 的任何人都能夠檢查 update 的有效性，並自行計算新的 $h' = \text{Hash}(x')$。

因為我們的 Merkle 證明本身就是「bags of cells」(參見 **2.5.10**)，可以將這樣的「增強 update」構造為一個「bag of cells」，其中包含 x 的舊根、其某些子孫以及從 x 的根到它們的路徑，以及 x' 的新根和其所有不屬於 x 的子孫。

2.5.13. Account state updates in a shardchain block. 特別是，在 shardchain block 中的賬戶狀態更新應按照 **2.5.12**中討論的方式進行增強。否則，某人可能提交一個包含在舊狀態中不存在的 cell 引用的無效狀態更新的 block；證明此 block 的無效性將是困難的（挑戰者如何證明 cell 不是先前狀態的一部分?)。

現在，如果包含在 block 中的所有狀態更新都是增強的，它們的有效性可以輕鬆檢查，並且其無效性也可以輕鬆顯示為違反（廣義）Merkle hashes 的遞迴定義特性。

2.5.14. "Everything is a bag of cells" philosophy. 前面的討論表明，我們在 TON Blockchain 或網絡中需要存儲或傳輸的所有內容都可以表示為「bag of cells」。這是 TON Blockchain 設計哲學的重要部分。一旦解釋了「bag of cells」方法並定義了一些「bags of cells」的「低階」序列化，就可以在抽象（依賴性）代數數據類型的高層次上定義所有內容（如 block 格式、shardchain 和賬戶狀態等)。

「一切都是 bag of cells」哲學的統一效果大大簡化了看似不相關的服務的實現；有關涉及付款通道的示例，請參見 **6.1.9**。

2.5.15. Block "headers" for TON blockchains. 通常，blockchain 中的 block 會從一個小型的 header 開始，其中包含前一個 block 的 hash、創建時間、block 中所有交易的樹的 Merkle hash 等。然後，block 的 hash 被定義為這個小型 block header 的 hash。因為 block header 最終取決於 block 中包含的所有數據，所以無法在不改變其 hash 的情況下修改 block。

在 TON blockchains 的 block 使用的 "bag of cells'' 方法中，沒有指定的 block header。相反，block hash 被定義為 block 的根 cell 的 (Merkle) hash。因此，block 的頂部（根）cell 可能被視為此 block 的小型 "header''。

但是，根 cell 可能不包含通常期望從這樣的 header 中獲得的所有數據。本質上，人們希望 header 包含 *Block* 數據類型中定義的某些字段。通常，這些字段將包含在幾個 cell 中，包括根 cell。這些 cell 一起構成了有關字段值的 "Merkle proof''。人們可能會堅持在 block 中的任何其他 cell 之前，

在一開始就包含這些 “header cells”。然後，只需要下載 block 序列化的前幾個字節，就可以獲得所有的 “header cells”，並了解所有期望的字段。

2.6 創建和驗證新的 Blocks

TON Blockchain 最終由 shardchain 和 masterchain blocks 組成。為了系統順利且正確地運作，必須創建、驗證這些 blocks，並通過網絡將其傳播到所有相關方。

2.6.1. Validators. 新的 blocks 由特定的節點創建和驗證，這些節點被稱為 *validators*。實質上，任何希望成為 validator 的節點都可以成為 validator，前提是它可以在 masterchain 中存入足夠多的抵押金（以 TON 幣，即 Grams；參考 Appendix **A**）。Validators 為良好的工作獲得一些「獎勵」，即所有交易、存儲和燃氣費用，以及一些新鑄造的幣，這反映了整個社群對 validators 的「感激」，因為它們使 TON Blockchain 持續運作。這筆收入按照所有參與 validators 的抵押金比例分配。

然而，成為 validator 是一項重大的責任。如果 validator 簽署了一個無效的 block，它可能會失去部分或全部的抵押金，並且可能會暫時或永久地被排除在 validators 之外。如果 validator 不參與創建 block，它不會收到與該 block 相關的獎勵部分。如果 validator 長時間不創建新的 blocks，它可能會失去部分的抵押金，並被暫停或永久排除在 validators 之外。

這一切都意味著 validator 不是輕而易舉地獲得金錢。事實上，它必須追踪所有或某些 shardchains 的狀態（每個 validator 負責驗證和創建某一子集 shardchains 中的新 blocks），執行這些 shardchains 中的智能合約所請求的所有計算，接收其他 shardchains 的更新等等。這項活動需要大量的磁碟空間、計算能力和網絡帶寬。

2.6.2. Validators instead of miners. 請記住，TON Blockchain 使用的是 Proof-of-Stake 方法，而不是 Bitcoin、當前版本的 Ethereum 和大多數其他加密貨幣採用的 Proof-of-Work 方法。這意味著人們不能通過提供某些工作證明（計算大量其他無用的 hashes）來「挖掘」新的 block，並因此獲得一些新的幣。相反，人們必須成為 validator，並花費自己的計算資源來存儲和處理 TON Blockchain 的請求和數據。簡而言之，要挖新幣，必須成為 *validator*。就這一點而言，*validators* 就是新的 *miners*。

但是，除了成為 validator 之外，還有一些其他方式可以賺取幣。

2.6.3. Nominators and “mining pools”. 要成為 validator，通常需要購買和安裝幾臺高性能的伺服器，並為它們提供良好的互聯網連接。這不像

當前挖掘比特幣所需的 ASIC 設備那麼昂貴。但你絕對不能在家用電腦上挖新的 TON 幣，更不用說智能手機了。

在比特幣、以太坊和其他 Proof-of-Work 加密貨幣挖掘社區中，有一個名為 *mining pools* 的概念，其中許多節點因計算能力不足而無法自行挖掘新的 blocks，所以他們合併力量並在之後分享獎勵。

Proof-of-Stake 世界中的相應概念是 *nominator*。實質上，這是一個將其資金借給 validator 以增加其抵押金的節點；然後 validator 將其獎勵的相應部分（或之前同意的一部分，例如 50%）分配給 nominator。

通過這種方式，nominator 也可以參與「挖掘」並獲得與其存款金額成正比的一些獎勵。它只獲得 validator 獎勵的一部分，因為它只提供了「資本」，但不需要購買計算能力、存儲和網絡帶寬。

但是，如果 validator 因無效行為而失去其抵押金，nominator 也會失去其抵押金的部分。在這種意義上，nominator 分擔風險。它必須明智地選擇其 nominated validator，否則可能會損失資金。在這種意義上，nominator 進行加權決策並使用其資金「投票」支持某些 validator。

另一方面，這種提名或借貸系統使人們能夠成為 validator，而無需首先投入大量金錢購買 Grams（TON 幣）。換句話說，它防止持有大量 Grams 的人壟斷 validator 的供應。

2.6.4. Fishermen: obtaining money by pointing out others' mistakes. 另一種不成為 validator 而獲得一些獎勵的方式是成為一名 *fisherman*。實質上，任何節點都可以通過在 masterchain 中存入少量資金成為 fisherman。然後，它可以使用特殊的 masterchain 交易來發布某些由 validators 之前簽名和發布的（通常是 shardchain）blocks 的（Merkle）無效性證明。如果其他 validators 同意這個無效性證明，則違規的 validators 將被懲罰（失去其抵押金的一部分），fisherman 則獲得一些獎勵（從違規的 validators 中沒收的幣的一部分）。之後，如 **2.1.17**中所述，必須更正無效的（shardchain）block。更正無效的 masterchain blocks 可能需要在先前提交的 masterchain blocks 之上創建「垂直」blocks（參見 **2.1.17**）；無需創建 masterchain 的分叉。

通常，一個 fisherman 需要成為至少某些 shardchains 的完整節點，並花費一些計算資源來運行至少一些智能合約的代碼。雖然 fisherman 不需要像 validator 那麼多的計算能力，但我們認為，一個天生的 fisherman 是一個準備處理新 blocks，但尚未被選為 validator 的候選者（例如，由於未能存入足夠大的抵押金）。

2.6.5. Collators: obtaining money by suggesting new blocks to validators. 另一種不成為 validator 但可以獲得一些獎勵的方法是成為一個 *collator*。這是一個節點，它為 validator 準備並建議新的 shardchain block

候選者，並使用從此 shardchain 的狀態和其他 (通常是相鄰的) shardchains 中取得的資料進行補充（collated），伴隨適當的 Merkle 證明。當需要從鄰近的 shardchains 轉發一些消息時，這是必要的。然後，validator 可以輕鬆檢查所提議的 block 候選者的有效性，無需下載此或其他 shardchains 的完整狀態。

由於 validator 需要提交新的 (collated) block 候選者以獲得一些（"mining"）獎勵，因此有理由支付一部分獎勵給願意提供適當 block 候選者的 collator。這樣，validator 可以避免觀察鄰近 shardchains 的狀態，將其外包給 collator。

然而，我們預期在系統的初始部署階段不會有單獨指定的 collators，因為所有 validators 都將能夠為自己充當 collators。

2.6.6. Collators or validators: obtaining money for including user transactions. 使用者可以向一些 collators 或 validators 打開 micropayment 通道，並支付少量的幣以換取在 shardchain 中包括他們的交易。

2.6.7. Global validator set election. 每月一次（實際上是每 2^{19} 個 masterchain blocks）選舉 "global" 的 validators 集合。此集合在一個月前確定並被全球知悉。

要成為 validator，節點必須將一些 TON 幣（Grams）轉入 masterchain，然後將它們發送到特定的智能合約作為其建議的抵押金 s。與抵押金一起發送的另一個參數是 $l \geq 1$，這是此節點願意接受的相對於最小可能值的最大驗證負載。還有一個 l 的全球上限（另一個可配置參數）L，例如說是 10。

然後，這個智能合約選舉 global 的 validator 集合，只需選擇最大的建議抵押金的前 T 個候選者並公布其身份。最初，validators 的總數是 $T = 100$；隨著負載的增加，我們預期它將增長到 1000。它是一個可配置參數（參見 **2.1.21**）。

每個 validator 的實際抵押金如下計算：如果前 T 個提議的抵押金是 $s_1 \geq s_2 \geq \cdots \geq s_T$，那麼第 i 個 validator 的實際抵押金設定為 $s'_i := \min(s_i, l_i \cdot s_T)$。這樣，$s'_i/s'_T \leq l_i$，所以第 i 個 validator 不會獲得超過 $l_i \leq L$ 倍最弱 validator 的負載（因為負載最終與抵押金成正比）。

然後，當選的 validators 可以撤回他們未使用的抵押金部分，$s_i - s'_i$。不成功的 validator 候選人可以撤回他們所有的建議抵押金。

每個 validator 發布其 *public signing key*，並不一定等於抵押金來源的帳戶的公鑰。[18]

validators 的抵押金被凍結，直到他們被選舉的時期結束，再加上一個月，以防新的爭議出現（例如，發現一個由這些 validators 簽名的無效 block）。

[18]對於每次 validator 選舉，生成和使用新的密鑰對是有意義的。

之後，將返回抵押金，以及在此期間鑄造的 validator 的幣份額和處理的交易的費用。

2.6.8. Election of validator "task groups". 整體的全域驗證者集合（其中每個驗證者都被視為具有與其股份相等的多重性 - 否則驗證者可能會被誘使承擔多個身份並在它們之間劃分其股份）只用於驗證新的 masterchain 區塊。shardchain 的區塊只由特定選擇的驗證者子集驗證，這些驗證者是從在**2.6.7**中描述的選擇的全域驗證者集合中選擇的。

這些驗證者「子集」或「任務組」，為每個 shard 定義，每小時（實際上，每 2^{10} 個 masterchain 區塊）旋轉一次，它們在一小時前就已知，因此每個驗證者都知道它將需要驗證哪些 shards，並可以為此做準備（例如，通過下載丟失的 shardchain 數據）。

用於為每個 shard (w, s) 選擇驗證者任務組的算法是確定性偽隨機的。它使用驗證者嵌入到每個 masterchain 區塊中的偽隨機數（通過使用閾值簽名生成的共識生成）來創建一個隨機種子，然後例如為每個驗證者計算 $\text{Hash}(\text{code}(w).\,\text{code}(s).\mathit{validator\_id}.\mathit{rand\_seed})$。然後按此 hash 的值對驗證者進行排序，並選擇第一個驗證者，以便至少具有總驗證者股份的 $20/T$ 並且由至少 5 個驗證者組成。

這種選擇可以由特殊的智能合約完成。在這種情況下，選擇算法將可以輕鬆升級，無需通過**2.1.21**中提到的投票機制進行硬分叉。迄今為止提到的所有其他「常數」（例如 2^{19}、2^{10}、T、20 和 5）也都是可配置的參數。

2.6.9. Rotating priority order on each task group. 在 shard 任務組的成員上有一個特定的「優先順序」，取決於先前 masterchain 區塊的 hash 和 (shardchain) 區塊序列號。此順序是通過生成並排序上述的某些 hash 來確定的。

當需要生成新的 shardchain 區塊時，通常選擇用於創建此區塊的 shard 任務組驗證者是根據此旋轉「優先順序」的第一名。如果它未能創建該區塊，第二或第三個驗證者也可能這樣做。基本上，他們都可以建議他們的區塊候選者，但是由具有最高優先權的驗證者建議的候選者應該作為 Byzantine Fault Tolerant (BFT) 共識協議的結果獲勝。

2.6.10. Propagation of shardchain block candidates. 因為 shardchain 任務組的成員資格提前一小時就已知，它們的成員可以利用這段時間建立一個專用的「shard 驗證者多播覆蓋網絡」，使用 TON Network 的一般機制 (cf. **4.2**)。當需要生成一個新的 shardchain 區塊時—通常在最近的 masterchain 區塊被傳播後的一兩秒鐘—每個人都知道誰有最高的優先權生成下一個區塊 (cf. **2.6.9**)。這個驗證者將創建一個新的整合的區塊候選者，無論是自己還是在整合者的幫助下 (cf. **2.6.5**)。驗證者必須檢查（驗證）此

區塊候選者（尤其是如果它是由某個整合者準備的）並用其（驗證者）私鑰簽名。然後，使用預先安排的多播覆蓋網絡將區塊候選者傳播到任務組的其餘部分 (該任務組根據**4.2**中的解釋創建自己的私有覆蓋網絡，然後使用**4.2.15**中描述的串流多播協議的版本來傳播區塊候選者)。

真正的 BFT 方式是使用拜占庭多播協議，例如 Honey Badger BFT 中使用的協議 [11]：通過一個 $(N, 2N/3)$-消除代碼對區塊候選者進行編碼，將結果數據的 $1/N$ 直接發送到組的每個成員，並期望他們將其數據的部分直接多播到組的所有其他成員。

然而，一個更快且更簡單的方法（參見**4.2.15**）是將區塊候選者分成一系列簽名的一千字節區塊（「chunks」），通過 Reed–Solomon 或泉代碼（如 RaptorQ code [9] [14]）擴充它們的序列，並開始傳輸 chunks 到「多播網格」（即覆蓋網絡）中的鄰居，期望他們將這些 chunks 進一步傳播。一旦驗證者獲得足夠的 chunks 來從中重建區塊候選者，它就會簽署一個確認收據並通過其鄰居將其傳播到整個組。然後，它的鄰居停止向它發送新的 chunks，但可能會繼續發送這些 chunks 的（原始）簽名，認為該節點可以通過應用 Reed–Solomon 或泉代碼自行生成後續的 chunks（擁有所有必要的數據），將它們與簽名組合起來，並傳播給還沒有準備好的鄰居。

如果在移除所有「壞」節點後「多播網格」（覆蓋網絡）仍保持連接（回想一下，允許最多三分之一的節點以拜占庭方式壞掉，即以任意惡意方式行為），則此算法將以最快的方式傳播區塊候選者。

不僅指定的高優先級區塊創建者可以將其區塊候選者多播到整個組。第二和第三優先順序的驗證者可能會開始多播他們的區塊候選者，無論是立即還是在未能從最高優先級的驗證者那裡接收到區塊候選者之後。但是，通常只有最大優先權的區塊候選者將被所有（實際上，至少由三分之二的任務組）驗證者簽名並作為新的 shardchain 區塊提交。

2.6.11. Validation of block candidates. 一旦 block candidate 被 validator 接收並檢查其起始 validator 的簽名，接收 validator 會檢查此 block candidate 的有效性，執行其中的所有交易並確保其結果與所聲稱的一致。從其他區塊鏈導入的所有消息都必須在 collated data 中有適當的 Merkle 證明，否則 block candidate 將被視為無效（並且，如果此證明被提交到 masterchain，則可能會懲罰已經簽署此 block candidate 的 validator）。另一方面，如果 block candidate 被認定為有效，接收 validator 會簽署它並將其簽名傳播給組中的其他 validator，可以通過「mesh multicast network」或直接的網絡消息。

我們想強調，*validator* 在檢查（*collated*）*block candidate* 的有效性時，不需要訪問此 *shardchain* 或相鄰 *shardchain* 的狀態。[19] 這使得驗證可以非

[19]一個可能的例外是相鄰 shardchain 的輸出隊列的狀態，因為在這種情況下，Merkle 證

常快速地進行（不需要磁盤訪問），並減輕了 validator 的計算和存儲負擔（尤其是如果他們願意接受外部 collators 的幫助來創建 block candidate）。

2.6.12. Election of the next block candidate. 一旦 block candidate 收集到 task group 中至少三分之二（按 stake）的 validator 的有效簽名，它就有資格被提交為下一個 shardchain block。運行一個 BFT 協議來達成對所選 block candidate 的共識（可能有多個提議），所有「好的」validator 都會優先選擇該輪中優先級最高的 block candidate。運行此協議的結果是，該 block 會被至少三分之二的 validator（按 stake）的簽名所增強。這些簽名不僅證明了所問 block 的有效性，還證明了它是由 BFT 協議選出的。之後，將 block（無 collated data）與這些簽名組合，以確定的方式序列化，然後通過網絡傳播給所有相關方。

2.6.13. Validators must keep the blocks they have signed. 在他們成為 task group 的成員期間，以及之後至少一小時（或者 2^{10} blocks），validator 預期會保留他們已簽署和提交的 block。如果未能向其他 validator 提供簽署的 block，可能會受到懲罰。

2.6.14. Propagating the headers and signatures of new shardchain blocks to all validators. Validator 將新生成的 shardchain block 的 headers 和簽名傳播到全局 set of validators，使用類似於為每個 task group 創建的 multicast mesh network。

2.6.15. Generation of new masterchain blocks. 在所有（或幾乎所有）新的 shardchain block 生成之後，可以生成一個新的 masterchain block。這個程序基本上與 shardchain block 相同（參見 **2.6.12**），不同之處在於所有 validator（或至少三分之二的 validator）都必須參與此過程。因為新的 shardchain block 的 headers 和簽名被傳播給所有的 validator，所以每個 shardchain 中最新的 block 的 hash 可以並且必須被包含在新的 masterchain block 中。一旦這些 hash 被提交到 masterchain block，外部觀察者和其他 shardchain 可以認為新的 shardchain block 已經提交並且是不可變的（參見 **2.1.13**）。

2.6.16. Validators must keep the state of masterchain. 所有的驗證者都預期要追踪 masterchain 的狀態，而不依賴於彙整的數據。這很重要，因為驗證者工作組的知識是從 masterchain 的狀態中衍生出來的。

2.6.17. Shardchain blocks are generated and propagated in parallel. 通常，每個驗證者都是幾個 shardchain 任務組的成員; 他們的數量（因此驗

明的大小可能會變得過大。

證者的負載）大約與驗證者的股份成正比。這意味著驗證者同時運行多個新的 shardchain 區塊生成協議。

2.6.18. Mitigation of block retention attacks. 因為所有驗證者在只看到其標頭和簽名後就將一個新的 shardchain 區塊的 hash 插入到 masterchain 中，因此存在一個小概率，即生成此區塊的驗證者會密謀並試圖避免發佈整個新區塊。這將導致鄰近 shardchains 的驗證者無法創建新的區塊，因為一旦它的 hash 被提交到 masterchain 中，他們必須至少知道新區塊的輸出消息隊列。

為了緩解這種情況，新區塊必須從其他一些驗證者（例如，鄰近 shardchains 的任務組聯合的三分之二）收集簽名，證明這些驗證者確實擁有此區塊的副本並且願意在需要時將它們發送給其他驗證者。只有在提供這些簽名之後，新區塊的 hash 才可能包含在 masterchain 中。

2.6.19. Masterchain blocks are generated later than shardchain blocks. Masterchain 區塊大約每五秒生成一次，就像 shardchain 區塊一樣。但是，雖然所有 shardchains 中的新區塊的生成基本上是同時進行的（通常由釋放新的 masterchain 區塊觸發），但新的 masterchain 區塊的生成被故意延遲，以允許在 masterchain 中包含新生成的 shardchain 區塊的 hashes。

2.6.20. Slow validators may receive lower rewards. 如果一個驗證者“慢”，它可能無法驗證新的區塊候選者，並且收集提交新區塊所需的三分之二的簽名可能不需要它的參與。在這種情況下，它將收到與此區塊相關的較低份額的獎勵。

這為驗證者提供了一個激勵，以優化他們的硬體、軟體和網路連接，以便盡可能快地處理用戶交易。

但是，如果驗證者在區塊提交之前未簽名，則其簽名可能包含在接下來的一個或多個區塊中，然後這部分獎勵（根據已生成多少區塊而呈指數下降---例如，如果驗證者延遲 k 區塊則為 0.9^k）仍將給予此驗證者。

2.6.21. “Depth” of validator signatures. 通常，當驗證者簽署一個區塊時，該簽名只證明了區塊的相對有效性：只要這個和其他 shardchains 的所有先前區塊都是有效的，這個區塊就是有效的。驗證者不能因為將先前區塊中提交的無效數據視為理所當然而受到懲罰。

然而，區塊的驗證者簽名有一個稱為“深度”的整數參數。如果它是非零的，那就意味著驗證者也宣稱先前指定數量的區塊的 (相對) 有效性。這是“慢”或“暫時離線”的驗證者趕上並簽署一些已經提交但未經他們簽名的區塊的方式。然後區塊獎勵的一部分仍將給予他們（參見 **2.6.20**）。

2.6.22. Validators are responsible for *relative* validity of signed shardchain blocks; absolute validity follows. 我們想再次強調，驗證者在 shardchain 區塊 B 上的簽名只證明了該區塊的相對有效性（或者如果簽名有“深度”d，也可能是 d 個先前的區塊的相對有效性，參見 **2.6.21**；但這不太影響下面的討論）。換句話說，驗證者聲稱 shardchain 的下一個狀態 s' 是通過應用在 **2.2.6**中描述的區塊評估函數 ev_block 從先前的狀態 s 獲得的：

$$s' = ev\_block(B)(s) \tag{24}$$

這樣，如果原始狀態 s 被證明為“不正確”（例如，由於先前區塊的無效性），那麼簽署了區塊 B 的驗證者不能被懲罰。漁夫（參見 **2.6.4**）只有在發現一個區塊是相對地無效時才會抱怨。PoS 系統作為一個整體努力使每個區塊都是相對地有效的，而不是遞迴地（或絕對地）有效的。然而，注意到，如果 *blockchain* 中的所有區塊都是相對有效的，那麼它們所有的和 *blockchain* 作為一個整體都是絕對有效的；使用對 blockchain 的長度的數學歸納法可以輕易地證明這一語句。通過這種方式，輕鬆可驗證的區塊的相對有效性的聲明一起證明了整個 blockchain 的更強大的絕對有效性。

注意，通過簽署一個區塊 B，驗證者聲稱該區塊給定原始狀態 s 是有效的（即， (24)的結果不是值 $\perp$，表示下一個狀態不能被計算）。因此，驗證者必須執行在評估 (24)期間訪問的原始狀態的 cell 的最小形式檢查。

例如，想像一個情境，期望從提交到區塊的交易中訪問的賬戶的原始餘額的 cell 被發現有零個原始字節，而不是期望的 8 或 16。然後，原始餘額簡單地不能從 cell 中檢索，並且在嘗試處理該區塊時會發生“未處理的異常”。在這種情況下，驗證者不應該簽署這樣的區塊，否則將受到懲罰。

2.6.23. Signing masterchain blocks. 與 masterchain 區塊的情況略有不同：簽署一個 masterchain 區塊，驗證者不僅聲明其相對有效性，而且還聲明所有先前區塊的相對有效性，直到這個驗證者承擔其責任的第一個區塊（但不再往回）。

2.6.24. The total number of validators. 驗證者被選舉的總數的上限 T（參見 **2.6.7**）在迄今為止描述的系統中，不能超過，例如，幾百或一千，因為所有驗證者都預計參與 BFT 共識協議來創建每個新的 masterchain 區塊，並且尚不清楚這樣的協議是否可以擴展到數千參與者。更重要的是，masterchain 區塊必須收集至少三分之二的所有驗證者（按權益）的簽名，並且這些簽名必須包含在新區塊中（否則系統中的所有其他節點都沒有理由信任新區塊而不自己驗證它）。如果必須在每個 masterchain 區塊中包括超過，例如，一千個驗證者簽名，這將意味著每個 masterchain 區塊中有更多的數據，所有完整節點都要存儲並通過網絡傳播，以及花費更多的處理

能力來檢查這些簽名（在 PoS 系統中，完整節點不需要自己驗證區塊，但他們需要相反地檢查驗證者的簽名）。

雖然將 T 限制為一千個驗證者對於 TON Blockchain 的部署的第一階段似乎足夠了，但必須為未來的增長提供條款，當 shardchains 的總數變得如此之大，以至於幾百個驗證者不足以處理所有的 shardchains。為此，我們引入了一個額外的可配置參數 $T' \leq T$（原本等於 T），並且只有前 T' 名被選舉的驗證者（按權益）預計創建和簽署新的 masterchain 區塊。

2.6.25. Decentralization of the system. 有人可能會懷疑，像 TON Blockchain 這樣的 Proof-of-Stake 系統，依賴 $T \approx 1000$ 的驗證者來創建所有 shardchain 和 masterchain 區塊，是否會變得“太集中”，與像 Bitcoin 或 Ethereum 這樣的傳統 Proof-of-Work 區塊鏈相反，其中每個人（原則上）都可能開採一個新區塊，沒有礦工總數的明確上限。

但是，像 Bitcoin 和 Ethereum 這樣的受歡迎的 Proof-of-Work 區塊鏈，目前需要大量的計算能力（高“hash rates”）以成功的概率開採新區塊。因此，新區塊的開採趨於集中在幾個大玩家手中，他們投資大量資金建立充滿為開採優化的定制硬體的數據中心；以及幾個大型的開採池手中，它們集中並協調了無法自己提供足夠“hash rate”的大量人的努力。

因此，到 2017 年，超過 75%的新 Ethereum 或 Bitcoin 區塊由少於十個礦工產生。實際上，兩個最大的 Ethereum 開採池共同生產了超過一半的所有新區塊！顯然，這樣的系統比依賴 $T \approx 1000$ 節點生產新區塊的系統更為集中。

人們還可能注意到，成為 TON Blockchain 驗證者所需的投資——即購買硬體（例如，幾個高性能服務器）和權益（如有必要，可以通過提名者池輕鬆收集；參見 **2.6.3**）——比成為成功的獨立 Bitcoin 或 Ethereum 礦工所需的要少。實際上， **2.6.7**的參數 L 將迫使提名者不加入最大的“開採池”（即，積累了最大權益的驗證者），而是尋找目前正在接受提名者資金的較小的驗證者，甚至創建新的驗證者，因為這將允許更高比例的驗證者的——並且也是提名者的——權益被使用，從而獲得更大的開採獎勵。這種方式，TON Proof-of-Stake 系統實際上鼓勵去中心化（創建和使用更多的驗證者）並懲罰集中化。

2.6.26. Relative reliability of a block. 區塊的 *(*相對*)* 可靠性簡單地說是已簽署此區塊的所有驗證者的總權益。換句話說，如果這個區塊被證明是無效的，某些行為者會失去的金額。如果有人關心的交易傳輸值低於區塊的可靠性，可以認為它們足夠安全。從這個意義上說，相對可靠性是外部觀察者可以對特定區塊的信任度的衡量。

注意，我們談論的是區塊的相對可靠性，因為它保證該區塊是有效的前提是前一個區塊和所有其他被參考的 *shardchains'* 區塊都是有效的（參

見 **2.6.22**）。

一個區塊的相對可靠性在提交後可能會增加——例如，當遲來的驗證者的簽名被添加時（參見 **2.6.21**）。另一方面，如果其中一個驗證者因與其他區塊相關的不當行為而失去部分或全部權益，區塊的相對可靠性可能減少。

2.6.27. "Strengthening" the blockchain. 提供驅使驗證器提高區塊相對可靠性的激勵是很重要的。這樣做的一種方法是為驗證器分配小量的獎勵，以在其他 shardchains 的區塊上添加簽名。即使是“即將成為”的驗證器，他們已存入的股份不足以進入按股份排名前 T 的驗證器，並被包括在全局驗證器集合中（參見 **2.6.7**），也可能參與這項活動（如果他們同意在失去選舉後保持其股份凍結，而不是撤回）。這樣的驗證器可能同時充當漁民（參見 **2.6.4**）：如果他們必須檢查某些區塊的有效性，他們也可以選擇報告無效的區塊並收集相關獎勵。

2.6.28. Recursive reliability of a block. 人們也可以定義一個區塊的 *recursive reliability*，這是其相對可靠性與其引用的所有區塊的遞歸可靠性之間的最小值（即，masterchain 區塊、先前的 shardchain 區塊和一些相鄰 shardchains 的區塊）。換句話說，如果該區塊被證明是無效的，無論是因為它本身無效還是因為它所依賴的某個區塊無效，至少有人會失去這筆錢。如果人們真的不確定是否信任區塊中的特定交易，人們應該計算這個區塊的 *recursive* 可靠性，而不僅僅是 *relative* 的可靠性。

當計算遞歸可靠性時，沒有必要回溯太遠，因為如果我們回溯太遠，我們會看到已經解凍並提取的驗證器簽名的區塊。無論如何，我們不允許驗證器自動重新考慮那些舊的區塊（即，創建超過兩個月的區塊，如果使用當前的可配置參數的值），並從中創建分叉或使用“垂直 blockchains”（參見 **2.1.17**）修正它們，即使它們被證明是無效的。我們假設兩個月的時期提供了充足的機會來檢測和報告任何無效的區塊，因此如果在這段時期內沒有挑戰某個區塊，那麼它不太可能受到挑戰。

2.6.29. Consequence of Proof-of-Stake for light nodes. TON Blockchain 採用的 Proof-of-Stake 方法的一個重要結果是，TON Blockchain 的輕型節點（運行輕型客戶端軟體）不需要下載所有 shardchain 或甚至 masterchain 區塊的「標頭」，以便能夠自行檢查完整節點提供給其的 Merkle 證明，作為對其查詢的回答。

實際上，由於最近的 shardchain 區塊 hash 已包含在 masterchain 區塊中，完整節點可以輕鬆提供一個 Merkle 證明，從已知的 masterchain 區塊的 hash 開始，說明給定的 shardchain 區塊是有效的。接下來，輕型節點只需要知道 masterchain 的第一個區塊（其中宣布了第一組驗證器），這個區塊（或至少其 hash）可能被內置到客戶端軟體中，並且每個月後只需一

個 masterchain 區塊，其中宣布了新當選的驗證器集合，因為這個區塊將由上一組驗證器簽署。從那時起，它可以獲得最近的幾個 masterchain 區塊，或至少他們的標頭和驗證器簽名，並使用它們作為檢查完整節點提供的 Merkle 證明的基礎。

2.7 分割和合併 Shardchains

TON Blockchain 最具特色和獨特的功能之一是，當負載過高時，它能夠自動將 shardchain 分割為兩部分，並在負載下降時將它們合併回來（參見 **2.1.10**）。由於其獨特性和對整個項目可擴展性的重要性，我們必須詳細討論它。

2.7.1. Shard configuration. 請回憶，在任何給定的時間點，每個工作鏈 w 都被分割成一個或多個 shardchains (w, s)（參見 **2.1.8**）。這些 shardchains 可以由一棵二進制樹的葉子表示，其根為 $(w, \emptyset)$，且每個非葉子節點 (w, s) 都有子節點 $(w, s.0)$ 和 $(w, s.1)$。這樣，屬於工作鏈 w 的每個賬戶都被分配到確切的一個 shard，而知道當前的 shardchain 配置的每個人都可以確定包含賬戶 *account_id* 的 shard (w, s)：它是二進制字符串 s 是 *account_id* 前綴的唯一 shard。

shard 配置—即，這個 *shard binary tree*，或給定 w 的所有活躍 (w, s) 的集合（對應於 shard binary tree 的葉子）—是 masterchain 狀態的一部分，且對於跟踪 masterchain 的每個人都可用。[20]

2.7.2. Most recent shard configuration and state. 回想一下，最近的 shardchain 區塊的 hashes 被包含在每個 masterchain 區塊中。這些 hashes 被組織成一個 shard binary tree（實際上，是每個 workchain 的一系列樹）。這樣，每個 masterchain 區塊都包含最近的 shard 配置。

2.7.3. Announcing and performing changes in the shard configuration. shard 配置可以通過兩種方式更改：要麼將 shard (w, s) *split* 為兩個 shards $(w, s.0)$ 和 $(w, s.1)$，要麼將兩個「sibling」shards $(w, s.0)$ 和 $(w, s.1)$ *merged* 為一個 shard (w, s)。

這些 split/merge 操作在事先（例如，2^6；這是一個可配置的參數）的區塊中被宣布，首先在相應的 shardchain 區塊的「標頭」中，然後在引用這些 shardchain 區塊的 masterchain 區塊中。這個提前公告是為了讓所有相關方為計劃的變更做好準備（例如，建立一個 overlay multicast network 來分發新創建的 shardchains 的新區塊，如 **4.2**所述）。然後，更改首先

[20] 實際上，shard 配置完全由最後的 masterchain 區塊確定；這簡化了獲取 shard 配置的訪問。

提交到 shardchain 區塊的（標頭），然後傳播到 masterchain 區塊。這樣，masterchain 區塊不僅定義了在其創建之前的最新 shard 配置，而且還定義了下一個立即的 shard 配置。

2.7.4. Validator task groups for new shardchains. 回憶一下，每個 shard，即每個 shardchain，通常都被分配一個驗證器的子集（一個 validator task group）專用於在相應的 shardchain 中創建和驗證新區塊（參見 **2.6.8**）。這些 task groups 被選為某段時間（大約一小時），並且在提前知道的時間（也大約是一小時），在此期間是不變的。[21]

但是，實際的 shard 配置可能會在此期間因為 split/merge 操作而發生變化。必須為新創建的 shards 分配 task groups。這是如此完成的：

注意，任何活躍的 shard (w, s) 要麼是某個唯一確定的原始 shard (w, s') 的後代，意味著 s' 是 s 的前綴，要麼它將是原始 shards (w, s') 的子樹的根，其中 s 將是每個 s' 的前綴。在第一種情況下，我們簡單地將原始 shard (w, s') 的 task group 作為新 shard (w, s) 的 task group。在後一種情況下，新 shard (w, s) 的 task group 將是所有原始 shards (w, s') 的 task groups 的聯集，這些 shards 是 shard tree 中的 (w, s) 的後代。

這樣，每個活躍的 shard (w, s) 都被分配了一個明確定義的驗證器子集（task group）。當一個 shard 被分割時，兩個子節點都繼承了原始 shard 的整個 task group。當兩個 shards 被合併時，它們的 task groups 也被合併。

任何追踪 masterchain 狀態的人都可以計算每個活躍 shard 的驗證器 task groups。

2.7.5. Limit on split/merge operations during the period of responsibility of original task groups. 最終，新的 shard 配置將被考慮到，並且新的專用驗證器子集（task groups）將自動分配給每個 shard。在此之前，必須對 split/merge 操作施加某種限制；否則，如果原始 shard 迅速分裂成 2^k 個新的 shards，則原始 task group 可能最終同時驗證 2^k shardchains，對於大的 k。

這是通過對 active shard 配置與原始 shard 配置（目前用於選擇 validator task groups 的配置）之間的距離施加限制來實現的。例如，可能要求從 active shard (w, s) 到原始 shard (w, s') 在 shard tree 中的距離，如果 s' 是 s 的前驅（即 s' 是 binary string s 的前綴），則不得超過 3，如果 s' 是 s 的後繼（即 s 是 s' 的前綴），則不得超過 2。否則，不允許進行 split 或 merge 操作。

大致上，對於給定的 validator task groups 集合的責任期間，一個 shard 可以被分裂（例如，三次）或合併（例如，兩次）的次數施加了限制。除此

[21] 除非某些驗證器因簽署無效的區塊而被臨時或永久禁止，那麼他們將自動從所有 task groups 中被排除。

之外，合併或分裂創建 shard 之後，某段時間（某些區塊數）內不能重新配置它。

2.7.6. Determining the necessity of split operations. shardchain 的 split 操作是由某些正式條件觸發的（例如，如果連續 64 個區塊的 shardchain 區塊至少有 90% 是滿的）。這些條件由 shardchain task group 監控。如果它們得到滿足，首先在新的 shardchain 區塊的標頭中包含一個「split 準備」標誌（並傳播到引用這個 shardchain 區塊的 masterchain 區塊）。然後，在幾個區塊之後，shardchain 區塊的標頭中包含「split 提交」標誌（並傳播到下一個 masterchain 區塊）。

2.7.7. Performing split operations. 在 shardchain (w, s) 的區塊 B 中包含「split commit」標誌後，該 shardchain 中不能有後續的區塊 B'。相反，將創建 shardchains $(w, s.0)$ 和 $(w, s.1)$ 的兩個區塊 B'_0 和 B'_1，分別參考區塊 B 作為它們的前一個區塊（並且它們都將通過標頭中的標誌指示 shard 剛剛被分裂）。下一個 masterchain 區塊將包含新 shardchains 的區塊 B'_0 和 B'_1 的 hashes；它不允許包含 shardchain (w, s) 的新區塊 B' 的 hash，因為「split commit」事件已經被提交到前一個 masterchain 區塊。

請注意，兩個新的 shardchains 將由與舊的 shardchain 相同的 validator task group 驗證，所以它們將自動擁有其狀態的副本。從 Infinite Sharding Paradigm 的角度來看，狀態分裂操作本身相對簡單（參見 **2.5.2**）。

2.7.8. Determining the necessity of merge operations. 合併 shard 操作的必要性也由某些正式條件檢測 (例如，如果連續 64 個區塊的兩個兄弟 shardchains 的區塊大小總和不超過最大區塊大小的 60%)。這些正式條件還應考慮這些區塊所消耗的總 gas，並將其與當前的區塊 gas 限制進行比較，否則由於有一些計算密集型的交易阻止了更多交易的納入，區塊可能會偶然變小。

這些條件由兩個兄弟 shards $(w, s.0)$ 和 $(w, s.1)$ 的 validator task groups 監控。請注意，兄弟節點在 hypercube 路由方面必然是鄰居 (參考 **2.4.19**)，因此任何 shard 的 task group 的 validators 都將在某種程度上監控兄弟 shard。

當滿足這些條件時，validator 子組之一可以通過發送特殊消息建議另一個合併。然後，它們組合成一個臨時的「合併任務組」，具有組合的成員資格，能夠運行 BFT 共識算法，並在必要時傳播區塊更新和區塊候選者。

如果他們就合併的必要性和準備情況達成共識，「merge prepare」標誌將提交到每個 shardchain 的一些區塊的頭部，並附帶至少三分之二的兄弟 task group 的 validators 的簽名（並被傳播到下一個 masterchain 區塊，以

便每個人都可以為即將到來的重新配置做好準備)。但是，他們繼續為一些預定義數量的區塊創建單獨的 shardchain 區塊。

2.7.9. Performing merge operations. 之後，當來自兩個原始 task groups 的聯合的 validators 準備成為已合併 shardchain 的 validators 時 (這可能涉及從兄弟 shardchain 的狀態轉移和一個狀態合併操作)，他們在其 shardchain 的區塊的頭部提交一個「merge commit」標誌 (這一事件傳播到下一個 masterchain 區塊)，並停止在單獨的 shardchains 中創建新的區塊 (一旦出現 merge commit 標誌，在單獨的 shardchains 中創建區塊是被禁止的)。相反，創建了一個已合併的 shardchain 區塊 (由兩個原始 task groups 的聯合創建)，在其「header」中參考它的兩個「preceding blocks」。這反映在下一個 masterchain 區塊中，該區塊將包含已合併 shardchain 的新創建區塊的 hash。之後，已合併的 task group 繼續在已合併的 shardchain 中創建區塊。

2.8 區塊鏈專案的分類

我們將通過將 TON 區塊鏈與現有和擬議的區塊鏈專案進行比較，來結束我們對 TON 區塊鏈的簡短討論。但在此之前，我們必須引入一個足夠通用的區塊鏈專案分類。基於此分類的特定區塊鏈專案的比較，將被推遲到 **2.9**。

2.8.1. Classification of blockchain projects. 作為第一步，我們建議一些用於區塊鏈（即，對於區塊鏈專案）的分類標準。任何這種分類都是有點不完整和表面的，因為它必須忽略正在考慮的專案的一些最具體和獨特的特點。然而，我們認為這是在提供至少一個非常粗略和大約的區塊鏈專案地圖的必要第一步。

我們考慮的標準列表如下：

- 單一區塊鏈與多區塊鏈架構 (參見 **2.8.2**)
- 共識算法：Proof-of-Stake 與 Proof-of-Work (參見 **2.8.3**)
- 對於 Proof-of-Stake 系統，使用的確切的區塊生成、驗證和共識算法（兩個主要選項是 DPOS 與 BFT; 參見 **2.8.4**）
- 對「任意的」(Turing-complete) 智能合約的支持 (參見 **2.8.6**)

多區塊鏈系統有額外的分類標準 (參見 **2.8.7**):

- 成員區塊鏈的類型和規則：同質的、異質的 (參見 **2.8.8**)，混合的 (參見 **2.8.9**)。聯邦 (參見 **2.8.10**)

- 有無主鏈，內部或外部 (參見 **2.8.11**)
- 對 sharding 的原生支持 (參見 **2.8.12**)。靜態或動態 sharding (參見 **2.8.13**)
- 成員區塊鏈之間的互動：鬆散聯接和緊密聯接的系統 (參見 **2.8.14**)

2.8.2. Single-blockchain vs. multi-blockchain projects. 第一個分類標準是系統中的區塊鏈數量。最古老和最簡單的專案由一個單一區塊鏈組成（簡稱「單鏈專案」）；更複雜的專案使用（或更確切地說，計劃使用）多個區塊鏈（「多鏈專案」）。

單鏈專案通常更簡單且經過更好的測試；它們經受住了時間的考驗。它們的主要缺點是低性能，或者至少是交易吞吐量，對於通用系統來說，這一數量在十 (Bitcoin) 到不到一百 (Ethereum) 的交易每秒。一些專用系統（如 Bitshares）能夠在區塊鏈狀態適合於記憶體的情況下，處理每秒數萬的專用交易，並將處理限制於一個預定義的特殊交易集，然後由像 C++ 這樣的語言編寫的高度優化的代碼執行（這裡沒有 VMs）。

多鏈專案提供了每個人都渴望的可擴展性。他們可能支持更大的總狀態和更多的每秒交易，但代價是使專案變得更為複雜，其實施更具挑戰性。因此，已經運行的多鏈專案很少，但大多數擬議的專案都是多鏈的。我們相信未來屬於多鏈專案。

2.8.3. 創建和驗證區塊: 工作量證明 **vs.** 權益證明. 另一個重要的區別是用於創建和傳播新區塊、檢查其有效性，以及在出現多個分支時選擇其中之一的算法和協議。

兩種最常見的範疇是 *Proof-of-Work (PoW)* 和 *Proof-of-Stake (PoS)*。工作量證明方法通常允許任何節點創建（“挖掘”）一個新區塊（並獲得與挖掘區塊相關的一些獎勵），前提是它有幸在其他競爭者成功之前解決一個否則無用的計算問題（通常涉及計算大量的 hashes）。在出現分支的情況下（例如，如果兩個節點發布兩個否則有效但不同的區塊來跟隨前一個），最長的分支勝出。這樣，區塊鏈的不變性保證是基於生成區塊鏈所花費的工作量（計算資源）：任何希望創建此區塊鏈的分支的人都需要重新做這些工作，以創建已提交區塊的替代版本。為此，一個人需要控制超過 50% 的創建新區塊所花費的總計算能力，否則替代分支變得最長的機會會指數性地降低。

權益證明方法基於一些特殊節點（驗證者）所做的大量賭注（以加密貨幣提名），以斷言它們已經檢查了一些區塊並發現它們是正確的。驗證者簽署區塊，並因此收到一些小獎勵；但是，如果一個驗證者被發現簽署了一個不正確的區塊，並且提供了這方面的證據，則其全部或部分賭注將被沒收。這樣，區塊鏈的有效性和不變性保證是由驗證者對區塊鏈有效性的總賭注給出的。

從這個角度看，權益證明更為自然，因為它激勵驗證者（它們取代了 PoW 礦工）執行有用的計算（需要檢查或創建新區塊，尤其是執行區塊中列出的所有交易），而不是計算其他無用的 hashes。這樣，驗證者會購買更適合處理用戶交易的硬件，以獲得與這些交易相關的獎勵，從整個系統的角度看，這似乎是一項相當有用的投資。

然而，權益證明系統在實施上有些挑戰，因為必須為許多罕見但可能的情況提供支援。例如，一些惡意的驗證者可能密謀破壞系統以獲取利益（例如，通過改變自己的加密貨幣餘額）。這導致了一些非常重要的博弈論問題。

簡而言之，權益證明更為自然且更有前景，尤其是對於多區塊鏈專案（因為如果有許多區塊鏈，工作量證明將需要過多的計算資源），但必須更加小心地考慮和實施。大多數目前運行的區塊鏈專案，尤其是最古老的專案（如 Bitcoin 和至少是原始的 Ethereum），使用工作量證明。

2.8.4. Variants of Proof-of-Stake. DPOS vs. BFT. 雖然 Proof-of-Work 算法彼此非常相似，主要差異在於必須計算以挖掘新區塊的 hash 函數，但 Proof-of-Stake 算法有更多的可能性。它們自己值得一個子分類。

基本上，人們必須回答關於 Proof-of-Stake 算法的以下問題：

- 誰可以產生（“挖掘”）一個新區塊 - 任何完整節點，或只是驗證者的（相對地）小子集的成員？(大多數 PoS 系統要求新區塊由數個指定的驗證者生成並簽名。)
- 驗證者是否通過他們的簽名保證區塊的有效性，還是所有完整節點都期望自己驗證所有區塊？(可擴展的 PoS 系統必須依賴驗證者的簽名，而不是要求所有節點驗證所有區塊鏈的所有區塊。)
- 是否有一個預先知道的指定生產者來生成下一個區塊鏈區塊，這樣其他人就不能代替它產生那個區塊？
- 新建的區塊最初只由一個驗證者（其生產者）簽署，還是它必須從一開始就收集大多數驗證者的簽名？

雖然似乎根據這些問題的答案有 2^4 可能的 PoS 算法類別，但在實踐中，區別主要歸結為兩種主要的 PoS 方法。事實上，大多數現代的 PoS 算法，旨在用於可擴展的多鏈系統，在前兩個問題上的答案是相同的：只有驗證者可以產生新區塊，並且他們保證區塊的有效性，而不要求所有完整節點自己檢查所有區塊的有效性。

至於最後兩個問題，它們的答案被證明是高度相關的，基本上只留下了兩個基本選項：

- *Delegated Proof-of-Stake (DPOS)*：每個區塊都有一個眾所周知的指定生產者；其他人不能生產該區塊；新區塊最初只由其生成的驗證者簽署。

- *Byzantine Fault Tolerant (BFT)* PoS 算法：有一個已知的驗證者子集，其中任何一個都可以建議一個新區塊；在多個建議的候選區塊中選擇實際的下一個區塊的選擇，必須在被發布到其他節點之前由大多數驗證者驗證並簽名，這是通過 Byzantine Fault Tolerant 共識協議的版本來實現的。

2.8.5. Comparison of DPOS and BFT PoS. BFT 方法的優勢是新產生的區塊從一開始就有大多數驗證者的簽名證明其有效性。另一個優點是，如果大多數驗證者正確執行 BFT 共識協議，則根本不會出現分叉。但另一方面，BFT 算法往往相當複雜，並且需要更多的時間讓驗證者子集達成共識。因此，區塊不能太頻繁地生成。這就是為什麼我們預期 TON Blockchain（從這個分類的角度看是一個 BFT 項目）每五秒只生成一個區塊。在實踐中，這個間隔可能會減少到 2-3 秒（儘管我們不承諾這一點），但如果驗證者分散在全球各地，則不會再減少。

DPOS 算法的優勢是相當簡單和直接。由於它依賴於預先知道的指定區塊生成者，所以可以非常頻繁地生成新區塊，例如，每兩秒一次，或者甚至每秒一次。[22]

然而，DPOS 要求所有節點 - 或至少所有驗證者 - 驗證收到的所有區塊，因為生成並簽署新區塊的驗證者不僅確認了此區塊的相對有效性，還確認了它引用的前一個區塊的有效性，以及在鏈中更遠的所有區塊（也許直到當前驗證者子集的責任期開始）。當前驗證者子集上有一個預定的順序，因此每個區塊都有一個指定的生產者（即，預期生成該區塊的驗證者）；這些指定的生產者按循環方式旋轉。這樣，一個區塊首先只由其生成的驗證者簽名；然後，當挖掘下一個區塊時，並且其生產者選擇引用此區塊而不是它的前一個區塊（否則它的區塊將位於較短的鏈中，這可能會在未來失去“最長分叉”競賽），下一個區塊的簽名本質上也是對前一個區塊的附加簽名。這樣，一個新區塊逐漸收集更多驗證者的簽名 - 例如，生成下一個區塊所需的時間內的二十個簽名。一個完整的節點將需要等待這二十個簽名，或者自己驗證區塊，從一個已充分確認的區塊開始（例如，回退二十個區塊），這可能並不容易。

DPOS 算法的明顯缺點是，只有在挖掘二十個更多的區塊後，一個新區塊（以及其內部的交易）才能達到相同的信任水平（“遞歸可靠性”如**2.6.28**中

[22]有些人甚至聲稱 DPOS 區塊生成時間為半秒，如果驗證者分散在幾個大陸上，這似乎不太現實。

所討論的），而 BFT 算法則立即提供這種信任水平（例如，二十個簽名）。另一個缺點是，DPOS 使用“最長分叉勝出”的方法來切換到其他分叉；如果至少有一些生產者在我們感興趣的那個之後無法生成後續的區塊（或者由於網絡分割或複雜的攻擊而未能觀察到這些區塊），這使得分叉相當可能。

我們認為，BFT 方法，雖然在實作上更複雜且產生區塊的間隔時間比 DPOS 長，但它更適合用於「緊密結合」(參考 **2.8.14**) 的多鏈系統。因為其他的區塊鏈可以在看到新區塊中的已提交交易（例如，生成給它們的消息）後幾乎立即開始行動，而不必等待 20 次確認有效性（即，接下來的 20 個區塊），或者等待接下來的六個區塊以確保沒有分叉出現並自己驗證新區塊（在可擴展的多鏈系統中，驗證其他區塊鏈的區塊可能變得過於繁重）。因此，他們可以在保持高度的可靠性和可用性（參考 **2.8.12**）的同時達到可擴展性。

另一方面，對於一個「鬆散結合」的多鏈系統，DPOS 可能是一個好選擇，即使區塊鏈之間不需要快速交互，例如，如果每個區塊鏈（「工作鏈」）代表一個單獨的分散式交換，且區塊鏈之間的互動限於從一個工作鏈到另一個的罕見代幣轉移（或者，更准確地說，以接近 1 : 1 的比率交易住在一個工作鏈中的一個 altcoin 和另一個）。這就是 BitShares 計畫實際上所做的事情，它非常成功地使用了 DPOS。

總的來說，雖然 DPOS 可以生成新的區塊並且將交易快速包含進它們（區塊間隔時間較短），但這些交易達到其他區塊鏈和 off-chain 應用程序所需的信任級別，作為「已提交」和「不可變」的，比在 BFT 系統中慢得多，例如，三十秒[23] 而不是五秒。更快的交易包含並不意味著更快的交易承諾。如果需要快速的區塊鏈間交互，這可能會成為一個巨大的問題。在這種情況下，人們必須放棄 DPOS，而選擇 BFT PoS。

2.8.6. Support for Turing-complete code in transactions, i.e., essentially arbitrary smart contracts. 區塊鏈項目通常在他們的區塊中收集一些 *transactions*，這些會以認為有用的方式更改區塊鏈的狀態（例如，從一個賬戶轉移某個加密貨幣金額到另一個賬戶）。有些區塊鏈項目可能只允許一些特定的預定義交易類型（如從一個賬戶到另一個賬戶的值轉移，提供正確的簽名）。其他人可能支持在交易中的一些有限的腳本形式。最後，一些區塊鏈支持在交易中執行任意複雜的代碼，使系統（至少原則上）能夠支持任意應用程序，只要系統的性能允許。這通常與「圖靈完整虛擬機和腳本語言」（意味著可以在任何其他計算語言中編寫的任何程序都可以重寫以在區塊鏈內部執行），以及「智能合約」（這些是住在區塊鏈中的程序）

[23]例如，EOS，迄今為止提出的最好的 DPOS 項目之一，承諾 45 秒的確認和區塊鏈間的交互延遲（參考 [5]，「交易確認」和「區塊鏈間通信的延遲」部分）。

相關聯。

當然，支持任意智能合約使系統真正靈活。另一方面，這種靈活性是有代價的：這些智能合約的代碼必須在某個虛擬機上執行，每次有人想創建或驗證一個區塊時，都必須對區塊中的每個交易執行這個操作。與可以通過在像 C++ 這樣的語言中實現它們的處理來優化的預定義和不可變的簡單交易類型的情況相比，這降低了系統的性能。

最終，對於任何通用區塊鏈項目，似乎都希望支持圖靈完整的智能合約; 否則，區塊鏈項目的設計者必須預先決定他們的區塊鏈將用於哪些應用程序。實際上，比特幣區塊鏈中對智能合約的支持不足是必須創建新的區塊鏈項目 Ethereum 的主要原因。

在一個（異質的; cf. **2.8.8**）多鏈系統中，人們可以通過在一些區塊鏈（即，工作鏈）中支持圖靈完整的智能合約，以及在其他區塊鏈中支持一小組高度優化的交易，來「兩全其美」。

2.8.7. Classification of multichain systems. 到目前為止，這個分類對單鏈和多鏈系統都是有效的。然而，多鏈系統承認幾個更多的分類標準，反映系統中不同的區塊鏈之間的關係。我們現在討論這些標準。

2.8.8. Blockchain types: homogeneous and heterogeneous systems. 在多鏈系統中，所有區塊鏈可能本質上都是相同的類型，並且有相同的規則（即，使用相同的交易格式、相同的虛擬機器執行智能合約代碼、共享相同的加密貨幣等），這種相似性被明確地利用，但每個區塊鏈中的數據都是不同的。在這種情況下，我們說該系統是 *homogeneous*。否則，不同的區塊鏈（在這種情況下通常被稱為 *workchains*）可以有不同的「規則」。然後我們說系統是 *heterogeneous*。

2.8.9. Mixed heterogeneous-homogeneous systems. 有時我們有一個混合系統，其中存在多個區塊鏈的類型或規則集，但許多具有相同規則的區塊鏈都存在，且這一事實被明確地利用。那麼它是一個混合的 *heterogeneous-homogeneous system*。據我們所知，TON Blockchain 是這種系統的唯一例子。

2.8.10. Heterogeneous systems with several workchains having the same rules, or *confederations*. 在某些情況下，具有相同規則的多個區塊鏈（workchains）可以存在於異質系統中，但它們之間的互動與具有不同規則的區塊鏈之間的互動相同（即，並未明確地利用其相似性）。即使他們似乎使用「相同的」加密貨幣，事實上他們使用不同的「altcoins」（加密貨幣的獨立化身）。有時，人們甚至可以使用接近 $1:1$ 的匯率轉換這些 altcoins。然而，我們認為這並不使系統變得均質；它仍然是異質的。我們說具有相同規則的這樣的異質 workchains 集合是一個 *confederation*。

雖然製作一個異質系統，允許創建具有相同規則的多個 workchains（即一個聯邦）可能看起來是建立可擴展系統的便宜方法，但這種方法也有很多缺點。從本質上講，如果有人在許多具有相同規則的 workchains 中托管一個大型項目，她得到的不是一個大型項目，而是該項目的許多小實例。這就像有一個聊天應用程序（或遊戲）只允許每個聊天（或遊戲）房間有最多 50 名成員，但「擴展」通過創建新房間在必要時容納更多用戶。結果，許多用戶可以參加聊天或遊戲，但我們能說這樣的系統真的可擴展嗎？

2.8.11. Presence of a masterchain, external or internal. 有時，一個多鏈計畫有一個特殊的「masterchain」（有時被稱為「控制區塊鏈」），用於存儲系統的整體配置（所有活躍的區塊鏈集合，或更準確地說是 workchains）、當前的驗證者集合（對於一個 Proof-of-Stake 系統）等。有時其他的區塊鏈被「綁定」到 masterchain，例如通過將它們最新的區塊的 hash 值提交到它（TON Blockchain 也是這麼做的）。

在某些情況下，masterchain 是 *external*，這意味著它不是該計畫的一部分，而是一個原先完全與其無關的區塊鏈，對於新計畫的使用並不知情。例如，人們可以嘗試使用 Ethereum 區塊鏈作為一個外部計畫的 masterchain，並為此目的在 Ethereum 區塊鏈上發布特殊的智能合約（例如，用於選舉和懲罰驗證者）。

2.8.12. Sharding support. 有些區塊鏈計畫（或系統）有原生的支持 *sharding*，這意味著幾個（必然是 homogeneous；參見 **2.8.8**）區塊鏈被視為單一（從高級角度看）虛擬區塊鏈的 *shards*。例如，人們可以創建 256 個具有相同規則的 shard 區塊鏈（「shardchains」），並根據其 *account_id* 的第一個字節來確定帳戶的狀態。

Sharding 是擴展區塊鏈系統的自然方法，因為，如果正確實施，系統中的用戶和智能合約根本不需要知道 sharding 的存在。事實上，當負載過高時，人們經常希望在現有的單鏈計畫（如 Ethereum）中添加 sharding。

另一種擴展方法是使用在 **2.8.10**中描述的異質 workchains 的「聯邦」，允許每個用戶在一個或多個她選擇的 workchains 中保留她的帳戶，並在必要時將資金從一個 workchain 轉移到另一個 workchain，基本上進行 $1:1$ 的 altcoin 交換操作。此方法的缺點已在 **2.8.10**中討論過。

但是，sharding 不容易在快速和可靠的方式中實施，因為它意味著不同的 shardchains 之間有很多消息。例如，如果帳戶在 N 個 shards 之間均勻分佈，且唯一的交易是從一個帳戶到另一個帳戶的簡單資金轉移，那麼所有交易的只有一小部分（$1/N$）將在單一的區塊鏈中執行；幾乎所有的（$1-1/N$）交易將涉及兩個區塊鏈，需要跨區塊鏈的通信。如果我們希望這些交易快速，我們需要一個用於在 shardchains 之間傳輸消息的快速系統。換句話說，區塊鏈計畫需要在 **2.8.14**中描述的意義上是「緊密結合的」。

2.8.13. Dynamic and static sharding. 分片可能是 *dynamic*（當需要時自動創建額外的分片）或是 *static*（有預先定義的分片數量，最好只能通過硬分叉來改變）。大多數分片提議都是靜態的；而 TON Blockchain 使用的是動態分片（參見 **2.7**）。

2.8.14. Interaction between blockchains: loosely-coupled and tightly-coupled systems. 多區塊鏈專案可以根據組件區塊鏈之間支持的互動水平進行分類。

最低的支持水平是不同區塊鏈之間完全沒有任何互動。我們在這裡不考慮這種情況，因為我們寧願說這些區塊鏈不是一個區塊鏈系統的部分，而只是相同區塊鏈協議的單獨實例。

下一個支持水平是缺乏對區塊鏈之間的消息傳送的任何具體支持，使得原則上可能進行互動，但很尷尬。我們稱這樣的系統為「鬆散耦合」；在這些系統中，人們必須像發送消息和轉移價值一樣，好像它們是屬於完全獨立的區塊鏈專案的區塊鏈（例如，比特幣和以太坊；想像兩方希望將存儲在比特幣區塊鏈中的比特幣兌換成存儲在以太坊區塊鏈中的以太）。換句話說，必須在源區塊鏈的區塊中包括出站消息（或其生成交易）。然後她（或其他某方）必須等待足夠的確認（例如，後續區塊的給定數量），以認為原始交易已被「提交」並「不可變」，從而能夠根據其存在執行外部操作。只有在此之後，才可以提交將消息中繼到目標區塊鏈的交易（也許還帶有源交易的參考和 Merkle 存在證明）。

如果在傳輸消息之前沒有等待足夠的時間，或者由於某些其他原因發生了分叉，那麼兩個區塊鏈的聯合狀態將被證明是不一致的：一條消息被遞送到第二個區塊鏈，而該消息從未在第一個區塊鏈中生成（最終選擇的分叉）。

有時會添加對消息傳送的部分支持，通過標准化所有工作鏈區塊中的消息格式和輸入和輸出消息隊列的位置（這在異構系統中尤其有用）。雖然這在某種程度上促進了消息傳遞，但在概念上與之前的情況沒有太大區別，所以這樣的系統仍然是「鬆散耦合」的。

相反，「緊密耦合」的系統包括特殊的機制，以提供所有區塊鏈之間的快速消息傳遞。期望的行為是能夠在生成源區塊鏈的區塊之後立即將消息遞送到另一個工作鏈。另一方面，「緊密耦合」的系統還應該在出現分叉的情況下維護整體的一致性。儘管這兩個要求乍看之下似乎是矛盾的，但我們相信 TON Blockchain 使用的機制（將 shardchain 區塊 hash 包含到 masterchain 區塊中；使用「垂直」區塊鏈來修復無效的區塊，參見 **2.1.17**；超立方體路由，參見 **2.4.19**；即時超立方體路由，參見 **2.4.20**）使其成為一個「緊密耦合」的系統，也許是迄今為止唯一的系統。

當然，建立一個「鬆散耦合」的系統要簡單得多；但是，快速和高效的

分片（參見 **2.8.12**）要求系統是「緊密耦合」的。

2.8.15. Simplified classification. Generations of blockchain projects. 我們到目前為止建議的分類方式將所有區塊鏈項目分割成多個類別。然而，我們使用的分類準則在實踐中確實具有很大的相關性。這使我們能夠提議一種簡化的「世代」方式來分類區塊鏈項目，作為現實的一個大略的估計，並給出一些例子。尚未被實施和部署的項目以 *italics* 表示；每一代的最重要特徵都用 **bold** 顯示。

- 第一代：單鏈，**PoW**，不支持智能合約。範例：Bitcoin (2009) 和許多其他不那麼有趣的仿造者（如 Litecoin、Monero 等）。
- 第二代：單鏈，PoW，支援智能合約。範例：Ethereum (2013; 2015 年部署)，至少在它的原始形式中。
- 第三代：單鏈，**PoS**，支援智能合約。範例：未來的 *Ethereum* (2018 或之後)。
- 替代的第三代 (3′)：多鏈，PoS，不支持智能合約，鬆散結合。範例：Bitshares (2013–2014; 使用 DPOS)。
- 第四代：多鏈，**PoS**，支持智能合約，鬆散結合。範例：*EOS* (2017; 使用 DPOS)，*PolkaDot* (2016; 使用 BFT)。
- 第五代：多鏈，使用 BFT 的 PoS，支援智能合約，緊密結合，具有分片功能。範例：*TON* (2017)。

儘管並非所有區塊鏈項目都確切地落入這些類別中，但大多數確實如此。

2.8.16. Complications of changing the "genome" of a blockchain project. 上述分類定義了區塊鏈項目的「基因組」。這個基因組非常「固定」：一旦項目部署並被許多人使用，就幾乎不可能更改它。更改它需要一系列的硬分叉（這需要社區的大部分人批准），即使如此，更改還需要非常保守，以保持向後兼容性（例如，更改虛擬機的語義可能會破壞現有的智能合約）。另一種方法是創建具有不同規則的新「側鏈」，並將它們以某種方式綁定到原始項目的區塊鏈（或區塊鏈）。人們可能使用現有單區塊鏈項目的區塊鏈作為本質上新的和獨立項目的外部主鏈。[24]

[24]例如，Plasma 項目計劃使用 Ethereum 區塊鏈作為其（外部）主鏈；它與 Ethereum 的其他部分互動不多，且可能由與 Ethereum 項目無關的團隊提議和實施。

我們的結論是，一旦部署了項目的基因組，就很難更改它。即使從 PoW 開始，並計劃在未來用 PoS 替換它，也相當複雜。[25] 為原始設計不支持它們的項目添加分片似乎幾乎不可能。[26] 實際上，將智能合約的支持添加到一個（即 Bitcoin）原始設計不支持這些功能的項目中被認為是不可能的（或至少被 Bitcoin 社區的大多數人認為是不受期望的），最終導致了一個新的區塊鏈項目，Ethereum。

2.8.17. Genome of the TON Blockchain. 因此，如果想建立一個可擴展的區塊鏈系統，必須從一開始就仔細選擇其基因組。如果系統在部署時未知但預期在未來支持一些特定功能，那麼它應該從一開始就支持「異質」工作鏈（可能具有不同的規則）。為了讓系統真正可擴展，它必須從一開始就支持分片；只有當系統是「緊密結合」時，分片才有意義（參考**2.8.14**），因此這反過來意味著存在主鏈，快速的區塊鏈間消息系統，BFT PoS 的使用等。

考慮到所有這些含義，為 TON 區塊鏈項目做出的大多數設計選擇似乎都是自然的，幾乎是唯一可能的選擇。

2.9 與其他區塊鏈項目的比較

我們試圖在包含現有和建議的區塊鏈項目的地圖上為其找到位置，從而總結對 TON 區塊鏈及其最重要和獨特功能的簡要討論。我們使用在**2.8**中描述的分類標準以統一的方式討論不同的區塊鏈項目，並建構這樣一個「區塊鏈項目地圖」。我們將此地圖表示為 Table 1，然後簡要地單獨討論幾個項目，以指出它們可能不適合一般計劃的特殊性。

2.9.1. Bitcoin [12]; `https://bitcoin.org/`. *Bitcoin* (2009) 是第一個且最知名的區塊鏈項目。它是典型的第一代區塊鏈項目：它是單一鏈，使用工作量證明 (Proof-of-Work) 與「最長分支勝出」的分支選擇算法，並且沒有圖靈完整的腳本語言（但是，支持沒有循環的簡單腳本）。Bitcoin 的區塊鏈沒有帳戶的概念；它使用 UTXO（未花費的交易輸出）模型。

2.9.2. Ethereum [2]; `https://ethereum.org/`. *Ethereum* (2015) 是第一個支持圖靈完整智能合約的區塊鏈。因此，它是典型的第二代項目，且是其中最受歡迎的。它在單一區塊鏈上使用工作量證明，但擁有智能合約和帳戶。

[25] 到 2017 年為止，Ethereum 仍在努力從 PoW 過渡到結合的 PoW+PoS 系統；我們希望它將來會成為一個真正的 PoS 系統。

[26] Ethereum 的分片提議可以追溯到 2015 年；目前還不清楚它們如何被實施和部署，而不會破壞 Ethereum 或創建一個本質上獨立的平行項目。

項目	年份	世代	共識	智能	鏈	類型	分片	交互
Bitcoin	2009	1	PoW	無	1			
Ethereum	2013, 2015	2	PoW	是	1			
NXT	2014	2+	PoS	無	1			
Tezos	2017, ?	2+	PoS	是	1			
Casper	2015, (2017)	3	PoW/PoS	是	1			
BitShares	2013, 2014	3′	DPoS	無	m	ht.	無	L
EOS	2016, (2018)	4	DPoS	是	m	ht.	無	L
PolkaDot	2016, (2019)	4	PoS BFT	是	m	ht.	無	L
Cosmos	2017, ?	4	PoS BFT	是	m	ht.	無	L
TON	2017, (2018)	5	PoS BFT	是	m	混合	dyn.	T

Table 1: 一些值得注意的區塊鏈項目的摘要。各列分別為：項目 —項目名稱；年份 —宣布年份和部署年份；世代 —世代 (參考**2.8.15**)；共識 —共識算法 (參考**2.8.3** 和**2.8.4**)；智能 —支持任意程式碼（智能合約；參考**2.8.6**)；鏈 —單/多區塊鏈系統 (參考**2.8.2**)；類型 —異質/同質多鏈系統 (參考**2.8.8**)；分片 —支持分片 (參考**2.8.12**)；交互 —區塊鏈間的交互，(L) 緩和或 (T) 緊密 (參考**2.8.14**)。

2.9.3. NXT; `https://nxtplatform.org/`. *NXT* (2014) 是第一個基於權益證明 (PoS) 的區塊鏈和貨幣。它仍然是單鏈，並且不支持智能合約。

2.9.4. Tezos; `https://www.tezos.com/`. *Tezos* (2018 或之後) 是一個建議的基於 PoS 的單一區塊鏈項目。我們在這裡提到它是因為它的獨特功能：其區塊解釋函數 *ev_block*（參考**2.2.6**）不是固定的，而是由一個 OCaml 模組決定，該模組可以通過在區塊鏈中提交一個新版本（並為所提議的更改收集一些票據）來升級。通過這種方式，一個人將能夠通過首先部署一個「原始」的 Tezos 區塊鏈，然後逐步地將區塊解釋函數更改為所需的方向，而無需硬分支。

這個想法，儘管引人入勝，但它有一個明顯的缺點，那就是它禁止在其他語言（如 C++）中的任何優化實現，因此基於 Tezos 的區塊鏈註定性能較低。我們認為，可能已經通過發布所提議的區塊解釋函數 *ev_trans* 的正式規範來獲得類似的結果，而不是確定特定的實現。

2.9.5. Casper.[27] *Casper* 是 Ethereum 的即將到來的 PoS 算法；如果在 2017 年（或 2018 年）的逐步部署成功，它將使 Ethereum 變成一個帶有智能合約支持的單鏈 PoS 或混合 PoW+PoS 系統，將 Ethereum 轉化為第三代項目。

2.9.6. BitShares [8]; `https://bitshares.org`. *BitShares* (2014) 是一個為分散式基於區塊鏈的交易所提供的平台。這是一個異構的多區塊鏈 DPoS

[27] `https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/`

系統，且不帶有智能合約；它透過僅允許一小部分預先定義的專用交易類型來實現其高性能，這些交易類型可以在 C++ 中有效地實現，前提是區塊鏈狀態適合於內存中。這也是第一個使用委託權益證明 (Delegated Proof-of-Stake, DPoS) 的區塊鏈項目，至少為某些專用目的證明其可行性。

2.9.7. EOS [5]; `https://eos.io`. *EOS* (2018 或以後) 是一個建議的異構多區塊鏈 DPoS 系統，並帶有智能合約支持和一些最小的消息傳遞支持（在**2.8.14**描述的意義上仍然是鬆散連接的）。這是先前成功創建了 BitShares 和 SteemIt 項目的同一團隊的嘗試，展示了 DPoS 共識算法的強大之處。通過為需要它的項目創建專用的工作鏈（例如，一個分散式交易所可能使用一個支持一組專用的優化交易的工作鏈，類似於 BitShares 所做的那樣）以及創建具有相同規則的多個工作鏈來實現可伸縮性（在**2.8.10**描述的意義上是聯邦）。這種可伸縮性方法的缺點和限制已在 *loc. cit.* 中討論。另見**2.8.5**、**2.8.12**和**2.8.14**，進一步討論 DPoS、分片、工作鏈之間的互動及其對區塊鏈系統的可伸縮性的影響。

與此同時，即使一個人不能在區塊鏈內「創建一個 Facebook」（參見**2.9.13**），不論是 EOS 還是其他方式，我們認為 EOS 可能會成為某些高度專用的弱互動分散式應用程序的便利平台，類似於 BitShares（去中心化交易所）和 SteemIt（去中心化博客平台）。

2.9.8. PolkaDot [17]; `https://polkadot.io/`. *PolkaDot* (2019 或之後) 是最佳思考且最詳細的建議多鏈權益證明項目之一；其開發是由 Ethereum 的共同創始人之一領導的。此項目是我們地圖上與 TON Blockchain 最接近的項目。（事實上，我們的「漁民」和「提名人」術語都是歸功於 PolkaDot 項目。）

PolkaDot 是一個異構的鬆散連接的多鏈權益證明項目，具有用於生成新區塊的拜占庭容錯（BFT）共識和主鏈（可能是外部的，例如 Ethereum 區塊鏈）。它還使用超立方體路由，有點像 TON 的慢速版本，如**2.4.19**所述。

其獨特的功能是它不僅可以創建公共的，而且還可以創建私人的區塊鏈。這些私人區塊鏈也將能夠與其他公共區塊鏈互動，不論是 PolkaDot 還是其他的。

因此，PolkaDot 可能會成為大規模私人區塊鏈的平台，例如，銀行聯盟可能會使用它來快速相互轉賬，或者大型公司可能對私有區塊鏈技術有其他用途。

然而，PolkaDot 不支持分片且不是緊密連接的。這在某種程度上限制了其可伸縮性，與 EOS 相似。（或許稍好一些，因為 PolkaDot 使用 BFT PoS 而非 DPoS。）

2.9.9. Universa; `https://universa.io`. 我們之所以在此提及這個不尋常的區塊鏈項目，是因為它是到目前為止唯一隨便提到與我們的 Infinite Sharding Paradigm 相似的項目（參見 **2.1.2**）。其另一個特點是它通過承諾只有項目的受信任和持有許可的夥伴才會被錄取為驗證者，因此他們永遠不會提交無效的區塊，來繞過所有與 Byzantine Fault Tolerance 相關的複雜性。這是一個有趣的決定；但是，它本質上使一個區塊鏈項目故意變得集中化，這是區塊鏈項目通常想要避免的（在一個受信任的集中化環境中為什麼需要區塊鏈?）。

2.9.10. Plasma; `https://plasma.io`). *Plasma* (2019?) 是 Ethereum 的另一位共同創始人的非傳統區塊鏈項目。它旨在緩解 Ethereum 的某些限制，而不引入 sharding。本質上，它是一個與 Ethereum 分開的項目，引入了一系列的（異構的）workchains，與最高級別的 Ethereum 區塊鏈綁定（作為一個外部 masterchain）。資金可以從層次結構中的任何區塊鏈轉移出去（從 Ethereum 區塊鏈作為根開始），並伴隨一個要完成的工作描述。然後在子 workchain 中進行必要的計算（可能需要將原始工作的部分進一步轉發到樹的下方），其結果被傳遞上去，並收集獎勵。通過一個（由 payment channel 啟發的）機制來避免實現這些 workchains 的一致性和驗證的問題，該機制允許用戶從一個行為不端的 workchain 單方面撤回他們的資金到其父 workchain（儘管很慢），並重新分配他們的資金和工作到另一個 workchain。

因此，Plasma 可能成為綁定到 Ethereum 區塊鏈的分散式計算平台，就像一個 ‘‘數學協處理器’’。但是，這似乎不像是實現真正的通用可伸縮性的方法。

2.9.11. Specialized blockchain projects. 還有一些專門的區塊鏈項目，如 FileCoin（一個激勵用戶為存儲願意支付費用的其他用戶的文件提供磁盤空間的系統），Golem（一個基於區塊鏈的平台，用於租用和出借計算能力給專門的應用，如 3D-rendering）或 SONM（另一個類似的計算能力出借項目）。這類項目在區塊鏈組織的層面上沒有引入任何概念上的新事物；相反，它們是特定的區塊鏈應用，可以由運行在通用目的區塊鏈中的智能合約實現，只要它可以提供所需的性能。因此，這類項目可能會使用現有或計劃中的區塊鏈項目作為其基礎，如 EOS、PolkaDot 或 TON。如果一個項目需要 ‘‘真正的’’ 可伸縮性（基於 sharding），那麼它最好使用 TON；如果它滿足於在一個 ‘‘聯邦’’ 背景下工作，通過為其目的明確定義一系列的 workchains，它可能會選擇 EOS 或 PolkaDot。

2.9.12. The TON Blockchain. TON（Telegram Open Network）Blockchain（計劃於 2018 年）是我們在本文檔中描述的項目。它旨在成為第一

個第五代區塊鏈項目——即 BFT PoS 多鏈項目，混合同質/異質性，支援（可 shardable）的自定義 workchains，具有原生的 sharding 支援，並且緊密結合（特別是，能夠在保持所有 shardchains 的一致狀態時，幾乎立即轉發 shards 之間的消息）。因此，它將是一個真正可伸縮的通用區塊鏈項目，能夠容納基本上可以在區塊鏈中實現的任何應用程序。當由 TON Project 的其他組件增強時（參見 **1**），其可能性甚至更大。

2.9.13. Is it possible to "upload Facebook into a blockchain"? 有時候人們聲稱，將 Facebook 這樣規模的社交網絡作為分佈式應用程序部署在區塊鏈中是可能的。通常會引用一個受喜愛的區塊鏈項目作為這種應用的可能"主機"。

我們不能說這是技術上的不可能。當然，需要一個緊密結合的區塊鏈項目，具有真正的 sharding（即 TON），以便這種大型應用不會運行得太慢（例如，從一個 shardchain 中的用戶傳送消息和更新到另一個 shardchain 中的朋友，且延遲合理）。然而，我們認為這是不需要的，且永遠不會被完成，因為價格會過高。

讓我們考慮將"將 Facebook 上載到區塊鏈"作為一個思考實驗；任何其他相似規模的項目也可能作為一個示例。一旦 Facebook 被上載到區塊鏈，目前由 Facebook 的伺服器完成的所有操作將被序列化為某些區塊鏈的交易（例如，TON 的 shardchains），並將由這些區塊鏈的所有驗證者執行。每項操作都必須執行，比如說，至少 20 次，如果我們希望每個區塊至少收集 20 個驗證者簽名（立即或最終，如在 DPOS 系統中）。同樣，Facebook 伺服器在其磁盤上保留的所有數據將被保留在相應 shardchain 的所有驗證者的磁盤上（即，至少有 20 份副本）。

因為驗證者基本上是與 Facebook 目前使用的相同伺服器（或許是伺服器集群，但這不影響此論點的有效性），我們可以看出，將 Facebook 運行在 blockchain 中的總硬件開銷至少比傳統方式實現高出 20 倍。

事實上，開銷還會更高，因為 blockchain 的虛擬機比運行優化編譯代碼的"裸 CPU"要慢，且其存儲未針對 Facebook 特定的問題進行優化。通過為 Facebook 設計具有某些特殊交易的特定 workchain，可以部分地減輕此問題；這是 BitShares 和 EOS 實現高性能的方法，也可在 TON Blockchain 中使用。然而，一般的 blockchain 設計本身仍然會帶來一些額外的限制，例如需要將所有操作註冊為 block 中的交易，將這些交易組織成 Merkle tree，計算和檢查它們的 Merkle hashes，進一步傳播這個 block 等。

因此，保守估計是，為了驗證承載該規模社交網絡的 blockchain 項目，需要的伺服器性能是 Facebook 現在使用的伺服器的 100 倍。有人將不得不為這些伺服器付錢，無論是擁有分散應用的公司（想像一下每個 Facebook 頁面上有 700 條廣告，而不是 7 條）還是它的用戶。無論哪種方式，這在

經濟上似乎都不可行。

我們認為，不是所有東西都應該上載到 *blockchain* 中。例如，不必在 blockchain 中保留用戶照片；將這些照片的 hashes 註冊到 blockchain 中，並將照片保存在分散的 off-chain 存儲（例如 FileCoin 或 TON Storage）中，可能是更好的選擇。這就是為什麼 TON 不僅僅是一個 blockchain 項目，而是圍繞 TON Blockchain 為中心的幾個組件的集合，如 Chapters **1**和 **5**所概述的。

3 TON 網路協議

任何 blockchain 項目不僅需要 block 格式和 blockchain 驗證規則的規範，還需要一個用於傳播新 block、發送和收集交易候選資料等的網路協議。換句話說，每個 blockchain 項目都必須設置一個專門的點對點網路。這個網路必須是點對點的，因為通常希望 blockchain 項目是分散式的，所以不能依賴於一組集中式的伺服器並使用傳統的客戶端-伺服器架構，例如，傳統的網路銀行應用程序所做的。即使是輕量級客戶端（例如，輕量級加密貨幣錢包智慧型手機應用程序），它們必須以客戶端-伺服器的方式連接到完整節點，如果先前的節點停止運作，實際上它們可以自由地連接到另一個完整節點，只要用於連接到完整節點的協議足夠標準化。

雖然如 Bitcoin 或 Ethereum 這樣的單一 blockchain 項目的網路需求可以很容易地被滿足（基本上需要構建一個“隨機”點對點的覆蓋網路，並通過一個 gossip 協議傳播所有新的 block 和交易候選資料），但像 TON Blockchain 這樣的多 blockchain 項目則要求更高（例如，人們必須能夠訂閱只有某些 shardchains 的更新，而不一定是所有的）。因此，TON Blockchain 和整個 TON 項目的網路部分至少值得簡短的討論。

另一方面，一旦需要支持 TON Blockchain 的更為複雜的網路協議到位，事實證明它們可以很容易地用於不一定與 TON Blockchain 的直接需求相關的目的，從而為 TON 生態系統中創建新服務提供了更多的可能性和靈活性。

3.1 抽象數據報網路層

建立 TON 網路協議的基石是 *(TON) Abstract (Datagram) Network Layer*。它使所有節點能夠假設某些“網路身份”，由 256 位“抽象網路地址”表示，並僅使用這些 256 位網路地址來識別發件人和收件人進行通信（首先互相發送數據報）。尤其是，人們不需要擔心 IPv4 或 IPv6 地址、UDP 端口號等；它們由 Abstract Network Layer 隱藏。

4 TON 網路協議

4.0.1. Abstract network addresses. 一個 *abstract network address*，或稱為 *abstract address*，或簡稱為 *address*，是一個 256 位整數，基本上等同於 256 位 ECC 公鑰。此公鑰可以任意生成，因此節點可以根據喜好創建許多不同的網路身份。但是，為了接收（和解密）針對此地址的消息，人們必須知道相對應的 *private* 密鑰。

實際上，地址不是公鑰本身；而是一個序列化 TL-object (參見 **2.2.5**) 的 256 位 hash（Hash = sha256），這可以根據其建構子（前四位元組）描述多種類型的公鑰和地址。在最簡單的情況下，此序列化 TL-object 僅由一個 4 位元組魔法數字和一個 256 位橢圓曲線密碼學（ECC）公鑰組成；在此情況下，地址將等於這 36 位元組結構的 hash。然而，人們也可以使用 2048 位 RSA 鑰匙或任何其他公鑰密碼學方案。

當一個節點獲知另一個節點的抽象地址時，它還必須接收其「原像」（即，其 hash 等於該抽象地址的序列化 TL-object）；否則，它將無法加密並發送數據報到該地址。

4.0.2. Lower-level networks. UDP implementation. 從幾乎所有 TON Networking 組件的角度看，唯一存在的就是一個網路（Abstract Datagram Networking Layer），能夠（不可靠地）從一個抽象地址發送數據報到另一個抽象地址。原則上，Abstract Datagram Networking Layer (ADNL) 可以在不同的現有網路技術上實施。但是，我們打算在 IPv4/IPv6 網路（如互聯網或內部網）上實施它，並在 UDP 不可用時選用 TCP 作為備選。

4.0.3. Simplest case of ADNL over UDP. 通過 UDP 從發送者的抽象地址發送數據報到任何其他抽象地址（已知原像）的最簡單情況可以如下實施。

假設發送者以某種方式知道擁有目標抽象地址的接收者的 IP 地址和 UDP 端口，且接收者和發送者都使用從 256 位 ECC 公鑰派生的抽象地址。

在此情況下，發送者簡單地通過其 ECC 簽名（用其私鑰完成）和其源地址（或源地址的原像，如果接收者還不知道該原像）增強要發送的數據報。結果使用收件人的公鑰加密，嵌入到一個 UDP 數據報中並發送到收件人已知的 IP 和端口。因為 UDP 數據報的前 256 位包含接收者的抽象地址，所以接收者可以確定應該使用哪個私鑰來解密數據報的其餘部分。只有在此之後，發送者的身份才被揭示。

4.0.4. Less secure way, with the sender's address in plaintext. 有時，一個較不安全的方案就足夠了，當收件人和寄件人的地址在 UDP 數

據報中保持為明文；寄件人的私鑰和收件人的公鑰使用 ECDH（橢圓曲線 Diffie–Hellman）合併在一起，生成一個 256 位共享秘密，之後，該共享秘密與明文部分也包含的隨機 256 位 nonce 一起用來派生用於加密的 AES 密鑰。完整性可以通過在加密前將原始明文數據的 hash 連接到明文來提供。

這種方法的優點是，如果預計兩個地址之間將交換多個數據報，則只需計算一次共享秘密然後將其緩存；然後加密或解密下一個數據報時不再需要較慢的橢圓曲線操作。

4.0.5. Channels and channel identifiers. 在最簡單的情況下，攜帶內嵌 TON ADNL 數據報的 UDP 數據報的前 256 位將等於收件人的地址。然而，一般來說它們組成了一個 *channel identifier*。有不同類型的 channels。其中一些是點對點的；它們由希望在將來交換大量數據的兩方創建，並通過交換幾個加密數據報（如**4.0.3**或**4.0.4**中所描述的那樣），運行經典或橢圓曲線 Diffie–Hellman（如果需要額外的安全性），或者僅由一方生成隨機共享秘密並將其發送給另一方。

此後，channel identifier 由共享秘密和一些額外數據（例如寄件人和收件人的地址）組合而來，例如通過 hashing，並且該 identifier 用作攜帶使用該共享秘密加密的數據的 UDP 數據報的前 256 位。

4.0.6. Channel as a tunnel identifier. 一般而言，“channel’’ 或 “channel identifier’’ 僅選擇了接收者已知的處理入站 UDP 數據報的方法。如果 channel 是接收者的抽象地址，則處理的方式如**4.0.3**或**4.0.4**中所述；如果 channel 是**4.0.5**中討論的已建立的點對點 channel，則處理包括使用共享秘密的幫助解密數據報，如 *loc. cit.* 中所解釋的，等等。

特別地，channel identifier 實際上可以選擇一個 “tunnel’’，當立即的接收者僅將接收到的消息轉發給其他人---實際的接收者或另一個代理時。沿途可能會進行一些加密或解密步驟（讓人想起 “onion routing” [6] 或甚至 “garlic routing”[28]），並且可能使用另一個 channel identifier 用於重新加密的轉發數據包（例如，點對點 channel 可以用於將數據包轉發給路徑上的下一個接收者）。

通過這種方式，可以在 TON Abstract Datagram Network Layer 的層次上添加對 “tunneling’’ 和 “proxying’’ 的支持---與 TOR 或 I^2P 項目提供的相似---而不影響所有高級 TON network protocols 的功能，它們將不知道這樣的添加。此機會被 *TON Proxy* 服務所利用（參見**5.1.11**）。

4.0.7. Zero channel and the bootstrap problem. 通常，一個 TON ADNL 節點會有一些 “neighbor table’’，其中包含有關其他已知節點的信

[28] `https://geti2p.net/en/docs/how/garlic-routing`

息，例如它們的抽象地址、它們的 preimages（即，公鑰）及它們的 IP 地址和 UDP 端口。然後，它將逐步擴展此表格，使用從這些已知節點作為特殊查詢的回答所學到的信息，並有時剪除過時的記錄。

但是，當一個 TON ADNL 節點剛啟動時，它可能不知道任何其他節點，只能學到一個節點的 IP 地址和 UDP 端口，但不知其抽象地址。例如，如果一個輕型客戶端無法訪問之前緩存的任何節點和硬編碼到軟件中的任何節點，並必須要求用戶輸入節點的 IP 地址或 DNS 域名，以通過 DNS 進行解析。

在這種情況下，節點將發送數據包到該節點的特殊 "zero channel''。這不需要知道收件人的公鑰（但消息仍應包含發件人的身份和簽名），所以消息是不加密地傳輸的。它通常只用於獲取接收者的身份（也許是特別為此目的創建的一次性身份），然後開始以更安全的方式通信。

一旦至少知道一個節點，就可以通過更多的條目輕鬆填充 "neighbor table'' 和 "routing table''，從已知節點發送的特殊查詢的回答中學到它們。

不是所有節點都需要處理發送到 zero channel 的數據包，但用於啟動輕型客戶端的節點應支持此功能。

4.0.8. TCP-like stream protocol over ADNL. ADNL，作為基於 256 位抽象地址的不可靠（小尺寸）數據包協議，可以用作更複雜網絡協議的基礎。例如，可以構建一個類似 TCP 的流協議，使用 ADNL 作為 IP 的抽象替代品。但是，TON Project 的大多數組件不需要這樣的流協議。

4.0.9. RLDP, or Reliable Large Datagram Protocol over ADNL. 一個建立在 ADNL 上的可靠的任意大小的數據包協議，稱為 RLDP，用於代替類似 TCP 的協議。這種可靠的數據包協議可以用來，例如，向遠程主機發送 RPC 查詢並從它們那裡接收答案（參見 **5.1.5**）。

4.1 TON DHT: Kademlia-like Distributed Hash Table

TON 分散式雜湊表 *(DHT)* 在 TON Project 的網絡部分中扮演了關鍵角色，用於定位網絡中的其他節點。例如，一個想要提交交易到某個 shardchain 的客戶端可能想要找到該 shardchain 的驗證者或 collator，或至少某個可以將客戶端的交易轉發到 collator 的節點。這可以通過在 TON DHT 中查找特定的 key 來完成。TON DHT 的另一個重要應用是，它可以快速填充新節點的鄰居表格 (參見 **4.0.7**)，只需查找隨機 key，或新節點的地址。如果節點對其入站數據報使用代理和通道，則將通道標識符和其入口點（例如，IP 地址和 UDP 端口）發布在 TON DHT 中；然後，所有希望發送數據報到該節點的節點首先將從 DHT 獲取此聯繫信息。

TON DHT 屬於 *Kademlia-like* 分散式雜湊表家族 [10]。

4.1.1. Keys of the TON DHT. TON DHT 的 *keys* 僅僅是 256 位整數。在大多數情況下，它們作為一個 TL-serialized 對象的 SHA256 計算出來（參見 **2.2.5**），被稱為 key 的 *preimage* 或 *key description*。在某些情況下，TON Network 節點的抽象地址（參見 **4.0.1**）也可以用作 TON DHT 的 keys，因為它們也是 256 位，而且它們也是 TL-serialized 對象的 hashes。例如，如果節點不害怕公開其 IP 地址，任何知道其抽象地址的人都可以通過在 DHT 中查找該地址作為 key 來找到它。

4.1.2. Values of the DHT. 分配給這些 256-bit keys 的 *values* 本質上是有限長度的任意字節串。對應 key 的 preimage 決定了這樣的字節串的解釋；通常由查找 key 的節點和存儲 key 的節點都知道它。

4.1.3. Nodes of the DHT. Semi-permanent network identities. TON DHT 的 key-value 映射存放在 DHT 的 *nodes* 上——基本上是 TON Network 的所有成員。為此，TON Network 的任何節點（除了某些非常輕的節點外），除了 **4.0.1**中描述的任意數量的瞬態和永久抽象地址外，都至少有一個“半永久地址”，該地址將其識別為 TON DHT 的成員。這個半永久或 *DHT* 地址不應該經常更改，否則其他節點將無法找到他們正在尋找的 keys。如果節點不想揭露其“真實”身份，則生成一個僅用於參與 DHT 的單獨抽象地址。但是，這個抽象地址必須是公開的，因為它將與節點的 IP 地址和端口相關聯。

4.1.4. Kademlia distance. 現在，我們既有 256 位的 keys 也有 256 位的 (半永久) 節點地址。我們引入所謂的 *XOR distance* 或 *Kademlia distance* d_K 在 256 位序列集上，由以下公式給出

$$d_K(x,y) := (x \oplus y) \quad \text{解釋為一個無符號的 256 位整數} \tag{25}$$

其中 $x \oplus y$ 表示兩個相同長度的位序列的位元對位元 eXclusive OR (XOR)。

Kademlia distance 在所有 256 位序列的集合 $\mathbf{2}^{256}$ 上引入了一個度量。特別地，我們有 $d_K(x,y) = 0$ 當且僅當 $x = y$，$d_K(x,y) = d_K(y,x)$，和 $d_K(x,z) \le d_K(x,y) + d_K(y,z)$。另一個重要特性是，從 x 出發的任何給定距離只有一個點：$d_K(x,y) = d_K(x,y')$ 意味著 $y = y'$。

4.1.5. Kademlia-like DHTs and the TON DHT. 如果一個擁有 256 位 keys 和 256 位節點地址的分散式雜湊表 (DHT) 預期將 key K 的值保留在與 K 最接近的 s 個 Kademlia 節點上 (即，與他們的地址到 K 的 Kademlia distance 最小的 s 個節點)，我們稱之為 *Kademlia-like DHT*。

這裡的 s 是一個小參數，比如說，$s = 7$，用於提高 DHT 的可靠性（如果我們只在一個節點上保留 key，即與 K 最接近的節點，那麼如果該節點離線，該 key 的值將會丟失）。

根據此定義，TON DHT 是一個 Kademlia-like DHT。它是在 **3.1**中描述的 ADNL 協議上實現的。

4.1.6. Kademlia routing table. 任何參與 Kademlia-like DHT 的節點通常都會維護一個 *Kademlia routing table*。對於 TON DHT，它由 $n = 256$ 個 buckets 組成，編號從 0 到 $n - 1$。第 i 個 bucket 將包含一些已知節點的資訊（一定數量 t 的 ‘‘最好的’’ 節點，和可能一些額外的候選節點），這些節點的 Kademlia distance 從 2^i 到 $2^{i+1} - 1$ 與節點地址 a 之間。[29] 這些資訊包括他們的 (半永久) 地址、IP 地址和 UDP 端口，以及一些可用性資訊，如最後一次 ping 的時間和延遲。

當一個 Kademlia 節點由於某個查詢得知其他 Kademlia 節點時，它將其包含到其路由表的適當 bucket 中，首先作為候選節點。然後，如果該 bucket 中的一些 ‘‘最好的’’ 節點失效（例如，長時間不回應 ping 查詢），它們可以被一些候選節點替換。通過這種方式，Kademlia routing table 保持填充。

Kademlia routing table 中的新節點也被包含在 **4.0.7**中描述的 ADNL 鄰居表中。如果 Kademlia routing table 的一個 bucket 中的 ‘‘最好的’’ 節點經常被使用，可以建立一個在 **4.0.5**中描述的 channel，以促進 datagrams 的加密。

TON DHT 的一個特殊特性是，它試圖選擇往返延遲最小的節點作為 Kademlia routing table 的 buckets 的 ‘‘最好的’’ 節點。

4.1.7. (Kademlia 網絡查詢。) 一個 Kademlia 節點通常支持以下網絡查詢：

- Ping – 檢查節點的可用性。
- Store$(key, value)$ – 要求節點保存 $value$ 作為 key 的值。對於 TON DHT，Store 查詢稍微複雜一些 (參考 **4.1.9**)。
- Find_Node(key, l) – 要求節點返回 l Kademlia 最接近的已知節點（從其 Kademlia 路由表中）到 key。
- Find_Value(key, l) – 如上所述，但如果節點知道對應於 key key 的值，則直接返回該值。

當任何節點想要查找 key K 的值時，它首先創建一組 s' 節點（對於某個小的 s' 值，例如，$s' = 5$），這些節點相對於所有已知節點的 Kademlia 距離最接近 K（即，它們是從 Kademlia 路由表中取出的）。然後，Find_Value

[29] 如果一個 bucket 中有足夠多的節點，它可以進一步細分為，比如說，八個子 bucket，這取決於 Kademlia distance 的前四位。這將加快 DHT 查找的速度。

查詢發送給它們中的每一個，並在其答案中提到的節點包含在 S 中。然後，S 中最接近 K 的 s' 節點也發送了 Find_Value 查詢（如果以前沒有這麼做），並且該過程繼續，直到找到該值或 S 停止增長。這是一種尋找相對於 Kademlia 距離最接近 K 的節點的 ‘‘光束搜索’’。

如果要設置某個 key K 的值，則為 $s' \geq s$ 運行相同的過程，使用 Find_Node 查詢而不是 Find_Value，以找到最接近 K 的 s 節點。之後，Store 查詢發送給它們所有人。

在 Kademlia 類似的 DHT 的實現中還有一些不太重要的細節（例如，任何節點應該每小時查找一次最接近自己的 s 節點，並通過 Store 查詢重新發布所有存儲的 key 給它們）。我們暫時忽略它們。

4.1.8. Booting a Kademlia node. 當一個 Kademlia 節點上線時，它首先通過查找自己的地址來填充其 Kademlia 路由表。在此過程中，它識別出最接近自己的 s 節點。它可以從它們那裡下載所有已知的 $(key, value)$ 對，以填充其 DHT 的部分。

4.1.9. Storing values in TON DHT. 在 TON DHT 中存儲值與一般的 Kademlia-like DHT 略有不同。當某人希望存儲一個值時，她不僅必須將鍵 K 本身提供給 Store 查詢，還必須提供其 *preimage*---也就是說，一個 TL 序列化的字符串（在開頭有幾個預定義的 TL-constructors）包含鍵的 ‘‘描述’’。此鍵描述稍後由節點保存，並與鍵和值一起保存。

鍵描述描述了存儲的對象的 ‘‘類型’’、其 ‘‘所有者’’ 以及在未來更新時的 ‘‘更新規則’’。所有者通常由包含在鍵描述中的公鑰識別。如果包含它，通常只接受由相應的私鑰簽名的更新。存儲的對象的 ‘‘類型’’ 通常只是一個字節字符串。但是，在某些情況下，它可以更為複雜---例如，輸入隧道描述（參考 **4.0.6**），或節點地址的集合。

‘‘更新規則’’ 也可能不同。在某些情況下，它們僅允許使用新值替換舊值，前提是新值由所有者簽名（簽名必須作為值的一部分保存，以便稍後由其他節點在獲得此鍵的值後進行檢查）。在其他情況下，舊值以某種方式影響新值。例如，它可以包含一個序列號，並且僅當新的序列號更大時才覆蓋舊值（以防止重放攻擊）。

4.1.10. Distributed “torrent trackers” and “network interest groups” in TON DHT. 另一個有趣的情況是當值包含一個節點列表---也許是它們的 IP 地址和端口，或者只是它們的抽象地址---和 ‘‘更新規則’’ 包括在此列表中包括請求者，只要她可以確認她的身份。

這種機制可以用於創建一個分佈式的 “torrent tracker’’，在其中所有對某個 “torrent’’（即，某個文件）感興趣的節點可以找到對同一個 torrent 感興趣或已經擁有副本的其他節點。

TON Storage (參考 **5.1.8**) 使用此技術來找到擁有所需文件副本的節點（例如，shardchain 的狀態快照或舊的區塊）。然而，其更重要的用途是創建 "overlay multicast subnetworks'' 和 "network interest groups''（參考 **4.2**）。其想法是只有一些節點對特定 shardchain 的更新感興趣。如果 shardchains 的數量變得非常大，則找到對相同 shard 感興趣的節點可能會變得很複雜。這個 "distributed torrent tracker'' 提供了一種方便的方法來找到這些節點。另一個選項是從驗證者那裡請求它們，但這不會是一個可擴展的方法，驗證者可能選擇不回應來自任意未知節點的這種查詢。

4.1.11. Fall-back keys. 目前描述的大多數 "key types'' 在其 TL 描述中都有一個額外的 32 位整數字段，通常等於零。但是，如果通過 hash 該描述獲得的鍵不能從 TON DHT 中檢索或更新，則會增加此字段中的值，並進行新的嘗試。這樣，通過創建許多靠近被攻擊鍵的抽象地址並控制相應的 DHT 節點，就不能 "capture'' 和 "censor'' 一個鍵（即，進行鍵保留攻擊）。

4.1.12. Locating services. 一些位於 TON Network 中的服務，通過（基於 TON ADNL 的）較高級別的協議（在 **3.1**中描述）提供，可能希望在某處公開其抽象地址，以便其客戶知道在哪裡找到它們。

但是，將服務的抽象地址發佈到 TON Blockchain 中可能不是最好的方法，因為可能需要經常更改抽象地址，並且可能有意義提供多個地址，出於可靠性或負載均衡的目的。

另一種方法是將公鑰發佈到 TON Blockchain 中，並使用一個特殊的 DHT 鍵，在 TL 描述字符串（參考 **2.2.5**）中指示該公鑰為其 "owner''，以發佈服務的抽象地址的最新列表。這是 TON Services 利用的方法之一。

4.1.13. Locating owners of TON blockchain accounts. 在大多數情況下，TON 區塊鏈賬戶的所有者不希望與抽象網絡地址，特別是 IP 地址相關聯，因為這可能侵犯他們的隱私。然而，在某些情況下，TON 區塊鏈賬戶的所有者可能想要發佈她可以被聯繫到的一個或多個抽象地址。

一個典型的情況是 TON Payments 的 "lightning network" 中的節點（參考 **6.2**），這是即時加密貨幣轉賬的平台。一個公開的 TON Payments 節點可能不僅想與其他節點建立支付通道，還想發佈一個抽象網絡地址，稍後可以用該地址聯繫它，以沿著已建立的通道進行支付。

一種選擇是在創建支付通道的智能合約中包含一個抽象網絡地址。更靈活的選擇是在智能合約中包括一個公鑰，然後像在 **4.1.12**中解釋的那樣使用 DHT。

最自然的方法是使用控制 TON 區塊鏈中賬戶的同一私鑰，來簽名和發佈關於與該賬戶相關的抽象地址的 TON DHT 中的更新。這幾乎與在 **4.1.12**中描述的方式相同; 但是，所使用的 DHT 鍵將需要一個特殊的鍵

描述，只包含 *account_id* 本身，等於“賬戶描述”的 sha256，其中包含賬戶的公鑰。該值中包含的簽名也將包含賬戶描述。

通過這種方式，提供了一種定位某些 TON 區塊鏈賬戶所有者的抽象網絡地址的機制。

4.1.14. Locating abstract addresses. 請注意，儘管 TON DHT 是在 TON ADNL 上實現的，但 TON ADNL 也用它來實現幾個目的。

其中最重要的是從其 256 位抽象地址開始定位節點或其聯繫數據。這是必要的，因為 TON ADNL 應該能夠向任意 256 位抽象地址發送數據包，即使沒有提供任何其他信息。

為此，只需將 256 位抽象地址作為 DHT 中的鍵查找。找到使用該地址的節點（即，使用此地址作為公共半永久 DHT 地址），在這種情況下，可以獲知其 IP 地址和端口; 或者，可以獲取一個由正確的私鑰簽名的問題鍵值的輸入隧道描述，在這種情況下，將使用此隧道描述將 ADNL 數據包發送到預期的接收者。

請注意，為了使抽象地址“公開”（可以從網絡中的任何節點到達），其所有者必須使用它作為半永久 DHT 地址，或在考慮的抽象地址下的 DHT 鍵中發佈一個輸入隧道描述，使用其另一個公共抽象地址（例如，半永久地址）作為隧道的入口點。另一個選擇是簡單地發佈其 IP 地址和 UDP 端口。

4.2 Overlay Networks 和 Multicasting Messages

在像 TON 區塊鏈這樣的多區塊鏈系統中，即使是全節點也通常只對獲取某些 shardchains 的更新（即新的區塊）感興趣。為此，必須在 TON Network 內部構建一個特殊的 overlay（子）網絡，基於在 **3.1**中討論的 ADNL 協議，每個 shardchain 一個。

因此，需要建立任意的 overlay 子網絡，對希望參與的任何節點開放。這些 overlay 網絡中將運行基於 ADNL 的特殊 gossip 協議。特別是，這些 gossip 協議可用於在這樣的子網絡內部傳播（廣播）任意數據。

4.2.1. Overlay networks. 一個 *overlay (sub)network* 只是在某個更大的網絡內部實現的（虛擬）網絡。通常只有較大網絡的一些節點參與 overlay 子網絡，並且只有這些節點之間的一些物理或虛擬“鏈接”是 overlay 子網絡的一部分。

這樣，如果將包含網絡表示為圖（在像 ADNL 這樣的數據包網絡的情況下可能是一個完整的圖，其中任何節點都可以輕鬆地與其他節點通信），則 overlay 子網絡是此圖的 *subgraph*。

在大多數情況下，使用構建於較大網絡的網絡協議之上的某些協議實現 overlay 網絡。它可以使用與較大網絡相同的地址，或使用自定義地址。

4.2.2. Overlay networks in TON. TON 中的 overlay 網絡基於在 **3.1**中討論的 ADNL 協議構建; 它們也使用 256 位 ADNL 抽象地址作為 overlay 網絡中的地址。每個節點通常選擇其抽象地址中的一個，也作為其在 overlay 網絡中的地址。

與 ADNL 相反，TON overlay 網絡通常不支持向任意其他節點發送數據包。相反，一些“半永久鏈接”在某些節點之間建立（被稱為考慮的 overlay 網絡的“鄰居”），並且消息通常沿這些鏈接轉發（即，從一個節點到其鄰居之一）。通過這種方式，TON overlay 網絡是 ADNL 網絡的（通常不完整的）子圖內。

TON overlay 網絡中的鄰居鏈接可以使用專用的點對點 ADNL 通道來實現（參考 **4.0.5**）。

overlay 網絡的每個節點都維護一個鄰居列表（關於 overlay 網絡），包含它們的抽象地址（它們用於在 overlay 網絡中識別它們）和一些鏈接數據（例如，用於與它們通信的 ADNL 通道）。

4.2.3. Private and public overlay networks. 有些 overlay 網絡是 *public*，意味著任何節點都可以隨意加入。另一些是 *private*，意味著只有某些節點可以被允許（例如，那些可以證明他們作為驗證者的身份的節點。）一些 private overlay 網絡甚至可能對“一般公眾”未知。這些 overlay 網絡的資訊只提供給某些受信任的節點；例如，它可以使用公鑰加密，並且只有擁有相應私鑰副本的節點才能解密此資訊。

4.2.4. Centrally controlled overlay networks. 有些 overlay 網絡是 *centrally controlled*，由一個或幾個節點或某個眾所周知的公鑰的所有者控制。其他則是 *decentralized*，意味著沒有特定節點負責它們。

4.2.5. Joining an overlay network. 當一個節點想要加入一個 overlay 網絡時，它首先必須學習其 256 位的 *network identifier*，通常等於 overlay 網絡的 *description* 的 SHA256—一個 TL-serialized 物件（參考 **2.2.5**），這可能包含 overlay 網絡的中央權威（即，其公鑰和可能的抽象地址[30]）一個與 overlay 網絡的名稱相對應的字符串，如果這是與該 shard 相關的 overlay 網絡，TON Blockchain shard 的標識符，等等。

有時從網絡標識符開始可以恢復 overlay 網絡描述，只需在 TON DHT 中查找它即可。在其他情況下（例如，對於 private overlay 網絡），必須與網絡標識符一起獲得網絡描述。

4.2.6. Locating one member of the overlay network. 在一個節點學到它想要加入的 overlay 網絡的網絡標識符和網絡描述之後，它必須定位屬

[30]或者，抽象地址可能存儲在 DHT 中，如 **4.1.12**所解釋的。

於該網絡的至少一個節點。

這也適用於不想加入 overlay 網絡，但只是想與其通信的節點；例如，可能有一個專用於為特定 shardchain 收集和傳播交易候選者的 overlay 網絡，客戶端可能想連接到此網絡的任何節點以建議交易。

用於定位 overlay 網絡成員的方法在該網絡的描述中定義。有時（尤其是對於 private 網絡），必須已經知道一個成員節點才能加入。在其他情況下，某些節點的抽象地址包含在網絡描述中。一種更靈活的方法是在網絡描述中只指示負責網絡的中央權威，然後抽象地址將通過某些 DHT 鍵的值提供，由該中央權威簽名。

最後，真正的 decentralized public overlay 網絡可以使用描述在 **4.1.10**中的“分佈式 torrent-tracker”機制，也使用 TON DHT 的幫助來實現。

4.2.7. Locating more members of the overlay network. Creating links. 一旦找到 overlay network 的一個節點，可以向該節點發送一個特殊的查詢，要求提供其他成員的列表，例如，被查詢節點的鄰居或其隨機選擇。

這使得加入的成員能夠根據 overlay network 填充其“相鄰性”或“鄰居列表”，通過選擇一些新學到的網絡節點並與它們建立連接（即，專用的 ADNL 點對點通道，如 **4.2.2**中所述）。之後，向所有鄰居發送特殊消息，表示新成員已準備好在 overlay network 中工作。鄰居在其鄰居列表中包含到新成員的連接。

4.2.8. Maintaining the neighbor list. Overlay network 節點必須不時更新其鄰居列表。一些鄰居，或至少是到它們的連接（通道），可能停止響應；在這種情況下，這些連接必須被標記為“暫停”，必須嘗試重新連接到這些鄰居，如果這些嘗試失敗，則必須銷毀這些連接。

另一方面，每個節點有時從隨機選擇的鄰居處請求其鄰居列表（或其隨機選擇），並使用它部分地更新自己的鄰居列表，通過向其添加一些新發現的節點，並移除一些舊的節點，無論是隨機的還是依賴於其響應時間和數據包丟失統計數據。

4.2.9. The overlay network is a random subgraph. 這樣，overlay network 在 ADNL 網絡內部成為一個隨機子圖。如果每個頂點的度至少為三（即，如果每個節點至少連接到三個鄰居），則這個隨機圖以接近一的概率被知道是 *connected*。更確切地說，具有 n 頂點的隨機圖是 *dis*connected 的概率是指數級小的，如果例如 $n \geq 20$，這個概率可以完全忽略。（當然，如果全球網絡分區時，這不適用，當分區的不同側面的節點沒有機會了解彼此時。）另一方面，如果 n 小於 20，只需要求每個頂點至少有，比如，至少十個鄰居。

4.2.10. TON overlay networks are optimized for lower latency. TON overlay 網絡按照以下方法優化前一方法生成的“隨機”網絡圖。每個節點都嘗試保留至少三個最小往返時間的鄰居，並很少更改這個“快速鄰居”列表。同時，它還具有至少其他三個完全隨機選擇的“慢鄰居”，以使 overlay network 圖始終包含一個隨機子圖。這是為了保持連接性並防止 overlay network 分裂為幾個未連接的區域子網絡。還選擇並保留至少三個“中間鄰居”，這些鄰居具有由某個常數（實際上，快鄰居和慢鄰居的往返時間的函數）界定的中間往返時間。

這樣，overlay network 的圖形仍然保持足夠的隨機性以保持連接，但是經過優化以實現較低的延遲和更高的吞吐量。

4.2.11. Gossip protocols in an overlay network. 在 overlay network 中，經常用來執行所謂的 *gossip protocols*，它在讓每個節點只與其鄰居互動的同時達到某個全局目標。例如，有一些 gossip protocols 用於構建一個 (不太大) overlay network 的所有成員的大致列表，或者只用每個節點有限的記憶體計算 (任意大) overlay network 的成員數量的估計 (參考 [15, 4.4.3] 或 [1] 了解詳情)。

4.2.12. Overlay network as a broadcast domain. 在 overlay network 中運行的最重要的 gossip protocol 是 *broadcast protocol*，旨在傳播由網絡的任何節點，或者其中一個指定的發送節點生成的廣播消息，給所有其他節點。

實際上有幾種廣播協議，為不同的使用情境進行了優化。其中最簡單的一種接收新的廣播消息，並將其轉發給尚未自己發送該消息副本的所有鄰居。

4.2.13. More sophisticated broadcast protocols. 某些應用程序可能需要更為複雜的廣播協議。例如，對於廣播大尺寸的消息，將新收到的消息的 hash（或新消息的 hash 集合）而不是消息本身發送給鄰居是有意義的。在學習到先前未見過的消息 hash 後，鄰居可以請求消息本身，例如，使用在 **4.0.9**中討論的可靠的大數據包協議 (RLDP) 進行傳輸。這樣，新消息只會從一個鄰居下載。

4.2.14. Checking the connectivity of an overlay network. 如果 overlay network 中有一個已知的節點（例如，overlay network 的“擁有者”或“創建者”），則可以檢查 overlay network 的連接性。然後，該節點不時地廣播包含當前時間、序列號和其簽名的短消息。任何其他節點如果在不久之前收到過這樣的廣播，就可以確定它仍然連接到 overlay network。此協議可以擴展到多個已知節點的情況；例如，它們都會發送這樣的廣播，所有其他節點都將期望從超過一半的已知節點那裡接收到廣播。

在用於傳播特定 shardchain 的新區塊（或僅新區塊頭）的 overlay network 的情況下，一個節點檢查連接性的好方法是追踪到目前為止收到的最新區塊。因為一個區塊通常每五秒生成一次，如果超過，比如，三十秒都沒有收到新的區塊，那麼節點可能已經從 overlay network 中斷開了。

4.2.15. Streaming broadcast protocol. 終於，TON overlay network 中有一個 *streaming broadcast protocol*，例如，用於在某些 shardchain 的驗證者之間傳播區塊候選者（"shardchain task group"），當然，他們為此目的創建了一個私有的 overlay network。相同的協議可以用來將新的 shardchain 區塊傳播給該 shardchain 的所有完整節點。

此協議已在 **2.6.10**中描述過：新的（大）廣播消息被分成，比如說，N 個一千字節的片段；這些片段的序列通過像 Reed-Solomon 或一個噴泉碼（例如，RaptorQ 碼 [9] [14]）這樣的消除碼擴展到 $M \geq N$ 的片段，並且這些 M 片段按升序的 chunk number 順序流向所有鄰居。參與的節點收集這些片段，直到它們可以恢復原始的大消息（為此，必須成功接收至少 N 的片段），然後指示其鄰居停止發送流的新片段，因為現在這些節點可以擁有原始消息的副本自己生成後續的片段。這些節點繼續生成流的後續片段並將它們發送給它們的鄰居，除非鄰居反過來指示不再需要這樣做。

這樣，節點在進一步傳播它之前不需要完整地下載一個大消息。這最大限度地減少了廣播延遲，尤其是當與 **4.2.10**中描述的優化相結合時。

4.2.16. Constructing new overlay networks based on existing ones. 有時，人們不想從頭開始構建一個 overlay network。相反，一個或幾個先前存在的 overlay network 是已知的，並且新的 overlay network 的成員資格預計會與這些 overlay network 的組合成員資格重疊。

一個重要的例子出現在 TON shardchain 被分成兩個，或兩個同級的 shardchains 合併為一（參見 **2.7**）。在第一種情況下，必須為每一個新的 shardchain 構建用於向完整節點傳播新區塊的 overlay networks；但是，可以預期這些新的 overlay network 中的每一個都包含在原始 shardchain 的區塊傳播網絡中（並包含大約一半的成員）。在第二種情況下，合併 shardchain 的新區塊的傳播的 overlay network 將大致由與正在合併的兩個同級 shardchains 相關的兩個 overlay network 的成員組成。

在這些情況下，新的 overlay network 的描述可能包含對與一系列相關的現有 overlay network 的明確或隱含的參考。希望加入新的 overlay network 的節點可以檢查它是否已經是這些現有網絡中的一個的成員，並詢問這些網絡中的鄰居是否也對新網絡感興趣。在得到肯定答案的情況下，可以建立到這些鄰居的新的點對點通道，並且它們可以被包括在新的 overlay network 的鄰居列表中。

這種機制並不完全取代在 **4.2.6**和**4.2.7**中描述的一般機制；相反，兩者都是並行運行的，並且用於填充鄰居列表。這是為了防止新的 overlay network 意外地分裂成幾個未連接的子網絡。

4.2.17. Overlay networks within overlay networks. 另一個有趣的案例出現在 *TON Payments* 的實現中（用於即時 off-chain 價值轉移的 "lightning network''；參見 **6.2**）。在這種情況下，首先構建包含所有 "lightning network'' 的轉發節點的 overlay network。但是，這些節點中的一些在區塊鏈中建立了支付通道；除了通過在 **4.2.6**、**4.2.7** 和 **4.2.8**中描述的一般 overlay network 算法選擇的任何 ''隨機'' 鄰居外，它們在此 overlay network 中必須始終是鄰居。這些與建立支付通道的鄰居的 ''永久鏈接'' 用於運行特定的 lightning network 協議，從而在包含的（幾乎始終連接的）overlay network 內部有效地創建了一個 overlay 子網絡（如果事情出錯，則不一定連接）。

5 TON 服務與應用程式

我們已經詳細討論了 TON 區塊鏈和 TON 網絡技術。現在我們將解釋它們如何可以組合起來創建各種服務和應用程式，並討論 TON 項目本身將提供的一些服務，無論是從一開始還是在稍後的時間。

5.1 TON 服務實施策略

我們首先討論如何在 TON 生態系統內實施不同的區塊鏈和網絡相關應用程式和服務。首先，一個簡單的分類是必要的：

5.1.1. Applications and services. 我們將 ‘‘應用程式’’ 和 ‘‘服務’’ 這兩個詞互換使用。但是，其中有一個微妙且有點模糊的區別：一個應用程式 通常直接向人類用戶提供一些服務，而一個服務 通常被其他應用程式和服務所利用。例如，TON Storage 是一個服務，因為它是代表其他應用程式和服務保存文件的，即使人類用戶也可能直接使用它。一個假設的 ‘‘在區塊鏈中的 Facebook” (參見 **2.9.13**) 或 Telegram 訊息傳遞應用，如果通過 TON 網絡提供 (即，作為一個 “ton-service’’ 實施；參見 **5.1.6**)，則更像是一個應用程式，即使一些 ‘‘機器人’’ 可能在沒有人工干預的情況下自動訪問它。

5.1.2. Location of the application: on-chain, off-chain or mixed. 為 TON 生態系設計的服務或應用程式需要將其數據保存並在某處處理該數據。這導致了以下應用程式（和服務）的分類：

- *On-chain* 應用程式 (參見 **5.1.4**)：所有數據和處理都在 TON 區塊鏈中。
- *Off-chain* 應用程式 (參見 **5.1.5**)：所有數據和處理都在 TON 區塊鏈之外，在通過 TON 網絡提供的服務器上。
- *Mixed* 應用程式 (參見 **5.1.7**)：部分，但不是所有，數據和處理都在 TON 區塊鏈中；其餘的都在通過 TON 網絡提供的 off-chain 服務器上。

5.1.3. Centralization: centralized and decentralized, or distributed, applications. 另一個分類標準是應用程式（或服務）是否依賴於集中的伺服器集群，或者真的是 ‘‘分佈式” (參見 **5.1.9**)。所有 on-chain 應用程式都自動地是去中心化和分佈式的。Off-chain 和混合應用程式可能呈現出不同程度的集中化。

現在讓我們更詳細地考慮上述可能性。

5.1.4. Pure "on-chain" applications: distributed applications, or "dapps", residing in the blockchain. 其中一種可能的方法,如在 **5.1.2**中提到的，是將一個「分散式應用程式」（通常縮寫為「dapp」）完全部署在 TON 區塊鏈中，作為一個智慧合約或一組智慧合約。所有數據都將作為這些智慧合約的永久狀態部分保存，並且所有與該項目的交互都將通過發送到或從這些智慧合約接收的 (TON 區塊鏈) 消息來完成。

我們已經在 **2.9.13**中討論過這種方法有其缺點和限制。它也有其優點：這樣的分散式應用程式不需要伺服器來運行或存儲其數據（它運行在「區塊鏈中」---即在驗證者的硬體上），並享有區塊鏈極高的 (拜占庭) 可靠性和可訪問性。這種分散式應用的開發者不需要購買或租用任何硬體；她需要做的只是開發一些軟體（即智慧合約的代碼）。之後，她將有效地從驗證者那裡租用計算能力，並用 Grams 支付，要麼由她自己支付，要麼由她的用戶承擔。

5.1.5. Pure network services: "ton-sites" and "ton-services". 另一個極端選擇是將服務部署在一些伺服器上，並通過在 **3.1**中描述的 ADNL 協議使其供用戶使用，也許還有一些更高級的協議，如在 **4.0.9**中討論的 RLDP，該協議可以用於以任何自訂格式向服務發送 RPC 查詢並獲得這些查詢的答案。這樣，該服務將完全在鏈外，並將駐留在 TON 網絡中，幾乎不使用 TON 區塊鏈。

TON 區塊鏈可能只用於查找服務的抽象地址或地址，如在 **4.1.12**中概述的，也許還可以利用像 TON DNS 這樣的服務（參見 **5.3.1**）來幫助將類似域的人類可讀字符串翻譯成抽象地址。

在 ADNL 網絡（即 TON 網絡）類似於 Invisible Internet Project（I^2P）的程度上，這樣的（幾乎）純網絡服務類似於所謂的「eep-services」（即，有一個 I^2P-地址作為他們的入口點的服務，並通過 I^2P 網絡提供給客戶）。我們會說，駐留在 TON 網絡中的這些純網絡服務是「ton-services」。

「eep-service」可以實現 HTTP 作為其客戶端-伺服器協議；在 TON 網絡上下文中，「ton-service」可能只是使用 RLDP（參見 **4.0.9**）數據報來傳輸 HTTP 查詢和響應。如果它使用 TON DNS 允許其抽象地址被一個人類可讀的域名查找，那麼與網站的類比就幾乎完美了。甚至可以寫一個專門的瀏覽器，或一個在用戶的機器上本地運行的特殊代理（「ton-proxy」），該代理接受來自用戶使用的普通網絡瀏覽器的任意 HTTP 查詢（一旦代理的本地 IP 地址和 TCP 端口輸入到瀏覽器的配置中），並將這些查詢通過 TON 網絡轉發到服務的抽象地址。然後用戶將擁有類似於世界寬網（WWW）的瀏覽體驗。

在 I^2P 生態系中，這樣的「eep-services」被稱為「eep-sites」。在 TON 生態系中也可以輕鬆創建「ton-sites」。這在某種程度上得到了 TON DNS

這類服務的幫助，它利用 TON 區塊鏈和 TON DHT 將 (TON) 域名翻譯成抽象地址。

5.1.6. Telegram Messenger as a ton-service; MTProto over RLDP. 我們想順便提到 MTProto 協議[31]，Telegram Messenger[32]用於客戶端-伺服器交互，可以輕鬆嵌入到在**4.0.9**中討論的 RLDP 協議，從而實質上將 Telegram 轉化為 ton-service。由於 TON Proxy 技術可以為 ton-site 或 ton-service 的終端用戶透明地開啟，並在 RLDP 和 ADNL 協議的下一級實施（參見**4.0.6**），這將使 Telegram 實質上不可封鎖。當然，其他消息和社交網絡服務也可能從這項技術中受益。

5.1.7. Mixed services: partly off-chain, partly on-chain. 一些服務可能使用混合方法：大部分處理在鏈外，但也有一些鏈上的部分（例如，為了登記他們對用戶的義務，反之亦然）。通過這種方式，狀態的一部分仍然會被保存在 TON Blockchain 中（即，不可變的公共分類帳），並且服務或其用戶的任何不當行為可以由智慧合約進行懲罰。

5.1.8. Example: keeping files off-chain; TON Storage. 這樣的服務的一個例子是由 TON Storage 提供的。在其最簡單的形式中，它允許用戶在鏈外存儲文件，只在鏈上保留要存儲的文件的哈希值，可能還有一個智慧合約，其中一些其他方同意在給定的時間段內為預先協商的費用保存該文件。實際上，文件可以被細分為一些小的大小（例如，1 千字節），由擦除代碼如 Reed–Solomon 或噴泉代碼進行增強，可以為增強的塊序列構建一個 Merkle tree 哈希值，並且這個 Merkle tree 哈希值可能在智慧合約中發布，而不是或與文件的通常哈希一起。這有點讓人想起文件在 torrent 中的存儲方式。

存儲文件的更簡單形式完全是在鏈外：人們可能基本上為新文件創建一個「torrent」，並使用 TON DHT 作為此 torrent 的「分散式 torrent 追踪器」（參見**4.1.10**）。這對於受歡迎的文件可能實際上工作得很好。但是，您不會得到任何可用性保證。例如，一個假設的「區塊鏈 Facebook」（參見**2.9.13**），它選擇在這種「torrents」中完全保留其用戶的個人資料照片，可能會冒著失去普通（不是特別受歡迎）的用戶的照片的風險，或者至少冒著在很長時間內無法顯示這些照片的風險。主要是在鏈外，但使用在鏈上的智慧合約來強制執行存儲的文件的可用性的 TON 存儲技術，可能更適合這項任務。

[31] `https://core.telegram.org/mtproto`
[32] `https://telegram.org/`

5.1.9. Decentralized mixed services, or "fog services". 到目前為止，我們已經討論了 *centralized* 混合服務和應用程式。雖然它們的鏈上組件是以去中心化和分散的方式進行處理的，位於區塊鏈中，但它們的鏈下組件依賴於某些由服務提供商以常見的集中方式控制的伺服器。而不是使用一些專用伺服器，計算能力可能會從其中一家大公司提供的雲計算服務中租用。但是，這不會導致服務的鏈下組件的去中心化。

實現服務的鏈下組件的去中心化方法在於創建一個市場，任何擁有所需硬體並願意出租其計算能力或磁碟空間的人都可以向需要它們的人提供服務。

例如，可能存在一個註冊表（也可以稱為「市場」或「交換所」），所有有興趣保存其他用戶文件的節點都在那裡發佈他們的聯繫信息，以及他們的可用存儲容量、可用性政策和價格。需要這些服務的人可能會在那裡查找它們，如果另一方同意，則在區塊鏈中創建智慧合約並上傳文件進行鏈下存儲。這樣，像 *TON Storage* 這樣的服務就真正成為去中心化的，因為它不需要依賴任何集中的伺服器集群來存儲文件。

5.1.10. Example: "fog computing" platforms as decentralized mixed services. 這種去中心化混合應用的另一個例子是當人們想執行一些特定計算（例如，3D 渲染或訓練神經網絡），通常需要特定且昂貴的硬體。然後擁有這種設備的人可能會通過類似的「交換所」提供他們的服務，需要這種服務的人則會租用它們，並通過智慧合約註冊雙方的義務。這類似於 "fog computing" 平台，如 Golem (`https://golem.network/`) 或 SONM (`https://sonm.io/`)，所承諾提供的。

5.1.11. Example: TON Proxy is a fog service. *TON Proxy* 提供了一個 fog 服務的另一個例子，希望提供其服務的節點（有償或無償）作為 ADNL 網絡流量的隧道可能會註冊，需要它們的人可能會根據提供的價格、延遲和帶寬選擇其中一個節點。之後，人們可能會使用由 *TON Payments* 提供的支付通道來處理代理服務的微支付，例如，每傳輸 128 KiB 收取的付款。

5.1.12. Example: TON Payments is a fog service. TON Payments 平台 (cf. **6**) 也是這種去中心化混合應用的一個例子。

5.2 連接使用者和服務提供者

我們在 **5.1.9**中看到「fog services」(即混合去中心化服務) 通常需要一些市場、交易所或註冊表，在那裡需要特定服務的人可以遇到提供這些服務的人。

這些市場很可能作為鏈上、鏈下或混合服務本身被實現，無論是集中式還是分散式。

5.2.1. Example: connecting to TON Payments. 例如，如果某人想使用 TON Payments(cf. **6**)，第一步是找到至少一些現有的「lightning network」(cf. **6.2**) 的轉接節點，並與它們建立支付通道，如果它們願意的話。可以使用「包圍」的覆蓋網絡找到一些節點，該網絡應該包含所有的轉接 lightning network 節點 (cf. **4.2.17**)。但是，這些節點是否願意創建新的支付通道還不清楚。因此，需要一個註冊表，那些準備創建新連接的節點可以在其中發佈他們的聯繫信息（例如，他們的抽象地址）。

5.2.2. Example: uploading a file into TON Storage. 同樣，如果某人想將文件上傳到 TON Storage，她必須找到一些願意簽署智慧合約的節點，將它們綁定到保留該文件的副本（或任何低於某個大小限制的文件）。因此，需要一個提供存儲文件服務的節點註冊表。

5.2.3. On-chain, mixed and off-chain registries. 這種服務提供者註冊表可能完全在鏈上實現，借助一個智慧合約來在其永久存儲中保留註冊表。但是，這將是相當緩慢和昂貴的。一種混合方法更為高效，其中相對較小且很少更改的鏈上註冊表只用於指出某些節點（通過它們的抽象地址，或者通過它們的公鑰，如在 **4.1.12**中描述的那樣，可以用來查找實際的抽象地址），這些節點提供鏈下（集中式）的註冊服務。

最後，一種去中心化的、純鏈下的方法可能包括一個公共的覆蓋網絡 (cf. **4.2**)，在那裡願意提供他們的服務的人，或者那些尋求購買某人服務的人，只需廣播由他們的私鑰簽名的優惠。如果要提供的服務非常簡單，甚至可能不需要廣播優惠：覆蓋網絡本身的大致成員資格可以用作願意提供特定服務的人的「註冊表」。然後，需要此服務的客戶可以定位 (cf. **4.2.7**) 並查詢此覆蓋網絡的一些節點，然後查詢其鄰居，如果已知的節點不準備滿足其需求。

5.2.4. Registry or exchange in a side-chain. 另一種實現去中心化混合註冊表的方法是創建一個獨立的專用區塊鏈（「side-chain」），由其自己的一組自稱為驗證者的人維護，他們在 on-chain 智能合約中發佈他們的身份，並為所有感興趣的方提供網絡訪問這個專用的區塊鏈，通過專用覆蓋網絡收集交易候選人和廣播塊更新（cf. **4.2**）。然後，這個 sidechain 的任何完整節點都可以維護其自己的共享註冊表副本（本質上等於此 side-chain 的全局狀態），並處理與此註冊表相關的任意查詢。

5.2.5. Registry or exchange in a workchain. 另一個選擇是在 TON 區塊鏈內創建一個專用的 workchain，專門用於創建註冊表、市場和交

易所。這可能比使用位於基本 workchain 中的智能合約更有效率和更便宜 (cf. **2.1.11**)。但是，這仍然比在 side-chains 中維護註冊表更昂貴 (cf. **5.2.4**)。

5.3 訪問 TON 服務

我們已經在 **5.1**中討論了創建居於 TON 生態系中的新服務和應用程序的不同方法。現在，我們討論如何訪問這些服務，以及 TON 將提供的一些「輔助服務」，包括 *TON DNS* 和 *TON Storage*。

5.3.1. TON DNS: a mostly on-chain hierarchical domain name service. *TON DNS* 是一個預定義的服務，它使用一系列的智能合約來從人類可讀的域名映射到 ADNL 網絡節點和 TON 區塊鏈賬戶和智能合約的（256 位）地址。

雖然原則上任何人都可以使用 TON 區塊鏈實現這樣的服務，但擁有這樣一個預定義的服務，並具有一個眾所周知的界面，當一個應用程序或服務希望將人類可讀的標識符轉換為地址時，將其用作默認選擇是很有用的。

5.3.2. TON DNS use cases. 例如，希望將一些加密貨幣轉移給另一個用戶或商家的用戶可能更喜歡記住該用戶或商家賬戶的 TON DNS 域名，而不是手頭保留他們的 256 位賬戶識別碼並將它們複製粘貼到他們的輕錢包客戶端的收件人字段中。

同樣地，TON DNS 可以用來查找智能合約的賬戶識別碼或 ton-services 和 ton-sites 的入口點（cf. **5.1.5**），使得一個專用客戶端（「ton-browser」）或一個與專用 ton-proxy 擴展或獨立應用程序結合的常見互聯網瀏覽器，能夠為用戶提供一個類似 WWW 的瀏覽體驗。

5.3.3. TON DNS smart contracts. TON DNS 是通過一系列特殊的 (DNS) 智能合約來實現的。每一個 DNS 智能合約都負責註冊某固定域的子域名。‘‘根” DNS 智能合約，即 TON DNS 系統的一級域名所在的位置，位於 masterchain 中。所有希望直接訪問 TON DNS 數據庫的軟件都必須將其帳戶標識符硬編碼。

任何 DNS 智能合約都包含一個 hashmap，將可變長度的以 null 結尾的 UTF-8 字符串映射到它們的 ‘‘值’’。這個 hashmap 是作為一個二進制的 Patricia 樹來實現的，與在 **2.3.7**中描述的類似，但支持可變長度的 bitstrings 作為鍵。

5.3.4. Values of the DNS hashmap, or TON DNS records. 至於值，它們是由 TL-scheme 描述的 “TON DNS 記錄” (cf. **2.2.5**)。它們包括一個 ‘‘魔數’’，選擇其中的一個支持的選項，然後是一個帳戶標識符、或一個智

能合約標識符、或一個抽象網絡地址 (cf. **3.1**)，或一個用於查找服務的抽象地址的公鑰 (cf. **4.1.12**)，或一個覆蓋網絡的描述，等等。一個重要的情況是另一個 DNS 智能合約：在這種情況下，該智能合約用於解析其域的子域名。通過這種方式，可以為不同的域創建單獨的註冊表，由這些域的所有者控制。

這些記錄還可以包含一個到期時間、一個緩存時間（通常非常大，因為在區塊鏈中太經常更新值是昂貴的），並且在大多數情況下引用所問子域名的所有者。所有者有權更改此記錄（尤其是所有者字段，從而將域名轉移給其他人的控制），並對其進行延長。

5.3.5. Registering new subdomains of existing domains. 為了註冊一個現有域的新子域，只需向該域的註冊者，即智能合約，發送一條消息，該消息包含要註冊的子域名（即鍵）、一個預定義格式之一的值、所有者的身份、一個到期日期，以及由該域的所有者確定的某個加密貨幣金額。

子域名是按照 ‘‘先到先得’’ 的原則註冊的。

5.3.6. Retrieving data from a DNS smart contract. 原則上，對於包含 DNS 智能合約的 masterchain 或 shardchain 的任何完整節點，只要知道智能合約的持久存儲中 hashmap 的結構和位置，就可以查找該智能合約數據庫中的任何子域。

然而，這種方法只適用於某些 DNS 智能合約。如果使用的是非標準的 DNS 智能合約，它將徹底失效。

相反，我們使用基於 *general smart contract interfaces* 和 *get methods*(cf. **5.3.11**) 的方法。任何 DNS 智能合約都必須定義一個帶有 ‘‘已知簽名’’ 的 “get method’’，用於查找一個鍵。由於這種方法也適用於其他智能合約，尤其是那些提供 on-chain 和混合服務的合約，我們在 **5.3.11**中詳細解釋了它。

5.3.7. Translating a TON DNS domain. 一旦任何完整節點，無論是獨立操作還是代表某個輕客戶端，都可以在任何 DNS 智能合約的數據庫中查找條目，就可以從已知的和固定的根 DNS 智能合約 (帳戶) 標識符遞歸地轉譯任意的 TON DNS 域名。

例如，如果想要轉譯 `A.B.C`，可以在根域數據庫中查找鍵`.C`、`.B.C` 和`A.B.C`。如果第一個鍵沒有找到，但第二個鍵找到了，且它的值是另一個 DNS 智能合約的引用，那麼就在該智能合約的數據庫中查找 `A`，並檢索最終值。

5.3.8. Translating TON DNS domains for light nodes. 這樣，對於 masterchain 的完整節點---以及參與域查找過程的所有 shardchains---都可

以在不需要外部幫助的情況下將任何域名轉譯為其當前值。輕節點可能會要求一個完整節點代替它執行這一操作，並返回值以及一個 Merkle 證明 (cf. **2.5.11**)。這個 Merkle 證明使得輕節點可以驗證答案是正確的，因此這樣的 TON DNS 響應不能被惡意攔截器 ‘‘偽造’’，這與通常的 DNS 協議形成了鮮明的對比。

因為不能期望任何節點都是對所有 shardchains 的完整節點，實際的 TON DNS 域轉譯將涉及這兩種策略的組合。

5.3.9. Dedicated “TON DNS servers”. 人們可以提供一個簡單的 “TON DNS 伺服器’’，它會接收 RPC “DNS” 查詢（例如，通過在 **3.1**中描述的 ADNL 或 RLDP 協議），請求伺服器轉譯給定的域名，如果需要，通過將某些子查詢轉發到其他（完整）節點來處理這些查詢，並返回對原始查詢的答案，如果需要，還可以增加 Merkle 證明。

這樣的 “DNS 伺服器’’ 可能會使用在 **5.2**中描述的其中一種方法，向任何其他節點，特別是輕型客戶端提供他們的服務（免費或付費）。請注意，如果這些伺服器被認為是 TON DNS 服務的一部分，它們將有效地將其從分佈式 on-chain 服務轉化為分佈式混合服務（即 “fog service’’）。

這結束了我們對 TON DNS 服務的簡短概述，這是一個可擴展的 on-chain 註冊表，用於 TON Blockchain 和 TON Network 實體的人類可讀域名。

5.3.10. Accessing data kept in smart contracts. 我們已經看到，有時需要訪問存儲在智能合約中的數據，而不改變其狀態。

如果知道智能合約實施的詳細信息，可以從智能合約的持久存儲中提取所有所需的信息，所有 shardchain 的完整節點都可以使用該智能合約。然而，這是一種相當不雅的做事方式，非常依賴於智能合約的實施。

5.3.11. “Get methods” of smart contracts. 一個更好的方法是在智能合約中定義一些 *get methods*，即一些類型的入站消息，當交付時不會影響智能合約的狀態，但會生成一個或多個包含 ‘‘結果’’ 的輸出消息的 get 方法。這樣，只要知道它實現了具有已知簽名的 get 方法（即已知要發送的入站消息的格式以及作為結果接收的出站消息的格式），就可以從智能合約中獲取數據。

這種方法更加優雅，並符合面向對象編程（OOP）。然而，到目前為止，它有一個明顯的缺陷：必須實際將交易提交到區塊鏈（將 get 消息發送到智能合約），等待它被提交並由驗證者處理，從新塊中提取答案，並支付燃氣費（即在驗證者的硬件上執行 get 方法）。這是資源的浪費：get 方法無論如何都不會改變智能合約的狀態，所以它們不需要在區塊鏈中執行。

5.3.12. Tentative execution of get methods of smart contracts. 我們已經提到 (cf. **2.4.6**) 任何完整節點都可以嘗試執行任何智能合約的任何方法（即，將任何消息交付給智能合約），從智能合約的給定狀態開始，而無需實際提交相應的交易。完整節點可以簡單地將正在考慮的智能合約的代碼加載到 TON VM 中，從 shardchain 的全局狀態（shardchain 的所有完整節點都知道）初始化其持久存儲，並使用入站消息作為其輸入參數執行智能合約代碼。所創建的輸出消息將產生此計算的結果。

通過這種方式，任何完整節點都可以評估任意智能合約的任意 get 方法，只要它們的簽名（即入站和出站消息的格式）是已知的。該節點可以跟踪在此評估期間訪問的 shardchain 狀態的 cell，並為可能要求完整節點這樣做的輕型節點創建 Merkle 證明，證明所執行的計算的有效性（cf. **2.5.11**）。

5.3.13. Smart-contract interfaces in TL-schemes. 回憶一下，由智能合約實現的方法（即它接受的輸入消息）基本上是一些 TL 序列化對象，這些對象可以由 TL-scheme 描述（cf. **2.2.5**）。所得到的輸出消息也可以由相同的 TL-scheme 描述。通過這種方式，智能合約提供給其他帳戶和智能合約的接口可以通過 TL-scheme 進行正式化。

特別是，可以通過這樣一個正式化的智能合約接口描述智能合約支持的（一部分）get 方法。

5.3.14. Public interfaces of a smart contract. 注意，一個正式化的智能合約接口，無論是以 TL-scheme 的形式（表示為 TL 源文件；cf. **2.2.5**）還是以序列化形式，都可以發布---例如，在區塊鏈中存儲的智能合約帳戶描述的特殊字段中，或者單獨發布，如果這個接口將被多次引用。在後一種情況下，支持的公共接口的 hash 可能會被合併到智能合約描述中，而不是接口描述本身。

這樣一個公共接口的例子是 DNS 智能合約的接口，它應該至少實現一個用於查找子域的標準 get 方法（cf. **5.3.6**）。在 DNS 智能合約的標準公共接口中也可以包括用於註冊新子域的標準方法。

5.3.15. User interface of a smart contract. 智能合約的公共介面存在還有其他好處。例如，當用戶請求查看一個智能合約時，一個錢包客戶端應用程序可以下載這樣的介面，並顯示智能合約支持的公共方法（即，可用的操作）列表，如果在正式介面中提供了一些人類可讀的評論，也許會顯示出來。在用戶選擇其中一種方法後，可以根據 TL-scheme 自動生成一個表單，在其中用戶將被提示選擇所需的所有字段和要附加到此請求的加密貨幣（例如，Grams）的所需金額。提交此表單將創建一個新的區塊鏈交易，其中包含剛剛組成的消息，從用戶的區塊鏈帳戶發送。

通過這種方式，只要這些智能合約已經發布了它們的介面，用戶將能夠通過填寫和提交某些表格以用戶友好的方式從錢包客戶端應用程序與任意智能合約進行互動。

5.3.16. User interface of a "ton-service". 事實證明，"ton-services"（即，居住在 TON Network 中並通過 **4**的 ADNL 和 RLDP 協議接受查詢的服務；cf. **5.1.5**）也可能從擁有公共介面中獲益，由 TL-schemes 描述（cf. **2.2.5**）。一個客戶端應用程序，例如一個輕型錢包或一個 "ton-browser"，可能會提示用戶選擇其中一個方法，並使用由介面定義的參數填寫一個表單，這與剛剛在 **5.3.15**中討論的類似。唯一的區別是，結果的 TL-serialized 消息不作為區塊鏈中的交易提交；相反，它作為一個 RPC 查詢發送到 "ton-service" 的抽象地址，並根據正式介面（即，一個 TL-scheme）解析和顯示此查詢的響應。

5.3.17. Locating user interfaces via TON DNS. 包含 ton-service 或智能合約帳戶標識符的抽象地址的 TON DNS 記錄還可能包含描述該實體的公共（用戶）介面的可選字段，或幾個支持的介面。然後，客戶端應用程序（無論是錢包、ton-browser 還是 ton-proxy）將能夠下載介面並以統一的方式與相應的實體（無論是智能合約還是 ton-service）進行互動。

5.3.18. Blurring the distinction between on-chain and off-chain services. 這樣，對於最終用戶，區塊鏈上、區塊鏈下和混合服務之間的區別變得模糊（cf. **5.1.2**）：她只需將所需服務的域名輸入到她的 ton-browser 或錢包的地址線中，其餘的工作由客戶端應用程序無縫處理。

5.3.19. A light wallet and TON entity explorer can be built into Telegram Messenger clients. 在此階段出現了一個有趣的機會。可以將實現上述功能的輕量錢包和 TON 實體瀏覽器嵌入到 Telegram Messenger 智能手機客戶端應用程序中，從而將技術帶給超過 2 億人。用戶可以通過在消息中包含 TON URIs（參見 **5.3.22**）來發送指向 TON 實體和資源的超連結；如果選擇了這些超連結，接收方的 Telegram 客戶端應用程序將在內部打開它，並開始與選定的實體互動。

5.3.20. "ton-sites" as ton-services supporting an HTTP interface. 一個 *ton-site* 簡單地說就是一個支持 HTTP 介面的 ton-service，也許還有其他一些介面。這種支持可以在相應的 TON DNS 記錄中宣布。

5.3.21. Hyperlinks. 請注意，由 ton-sites 返回的 HTML 頁面可能包含 *ton-hyperlinks* — 也就是，通過特殊製作的 URI 方案（參見 **5.3.22**）引用其他 ton-sites、智能合約和帳戶的參考 — 包含抽象網絡地址、帳戶識

別符或可讀的 TON DNS 域。然後，當用戶選擇它時，“ton-browser’’ 可能會遵循這樣的超連結，檢測要使用的介面，並顯示一個用戶介面表單，如**5.3.15**和 **5.3.16**中所述。

5.3.22. Hyperlink URLs may specify some parameters. 超連結 URL 可能不僅包含所述服務的 (TON) DNS 域或抽象地址，還包含要調用的方法名稱以及部分或所有的參數。這種 URI 方案可能如下所示：

`ton://`*<domain>*`/`*<method>*`?`*<field1>*`=`*<value1>*`&`*<field2>*`=...`

當用戶在 ton-browser 中選擇這樣的鏈接時，要么立即執行該操作（特別是如果它是匿名調用的智能合約的 get 方法），要么顯示一個部分填充的表格，由用戶明確確認和提交（對於支付表格可能需要這樣做）。

5.3.23. POST actions. ton-site 可以將它返回的 HTML 頁面嵌入到一些常見的 POST 表格中，其中的 POST 操作通過適當的 (TON) URLs 引用 ton-sites、ton-services 或智能合約。在這種情況下，一旦用戶填寫並提交了該自定義表單，就會採取相應的操作，無論是立即還是在明確確認後。

5.3.24. TON WWW. 上述所有內容都將導致在 TON Network 中建立一整個相互參照的實體網絡，終端用戶可以通過 ton-browser 訪問它，為用戶提供類似 WWW 的瀏覽體驗。對於終端用戶來說，這將最終使區塊鏈應用基本上與他們已經習慣的網站相似。

5.3.25. Advantages of TON WWW. 這個包括 on-chain 和 off-chain 服務的 “TON WWW’’ 相比於其常規對應物具有一些優勢。例如，支付在系統中是固有集成的。用戶身份可以始終呈現給服務（通過自動生成的交易和生成的 RPC 請求上的簽名），或者可以隱藏。服務不需要檢查和重新檢查用戶憑證；這些憑證可以一勞永逸地在區塊鏈中發布。可以很容易地通過 TON Proxy 保留用戶網絡匿名性，並且所有服務都將有效地無法被封鎖。由於 ton-browsers 可以與 TON 支付系統集成，所以也可能並且容易進行小額支付。

6 TON Payments

The last component of the TON Project we will briefly discuss in this text is *TON Payments*, the platform for (micro)payment channels and "lightning network" value transfers. It would enable "instant" payments, without the need to commit all transactions into the blockchain, pay the associated transaction fees (e.g., for the gas consumed), and wait five seconds until the block containing the transactions in question is confirmed.

The overall overhead of such instant payments is so small that one can use them for micropayments. For example, a TON file-storing service might charge the user for every 128 KiB of downloaded data, or a paid TON Proxy might require some tiny micropayment for every 128 KiB of traffic relayed.

While *TON Payments* is likely to be released later than the core components of the TON Project, some considerations need to be made at the very beginning. For example, the TON Virtual Machine (TON VM; cf. **2.1.20**), used to execute the code of TON Blockchain smart contracts, must support some special operations with Merkle proofs. If such support is not present in the original design, adding it at a later stage might become problematic (cf. **2.8.16**). We will see, however, that the TON VM comes with natural support for "smart" payment channels (cf. **6.1.9**) out of the box.

6.1 Payment Channels

We start with a discussion of point-to-point payment channels, and how they can be implemented in the TON Blockchain.

6.1.1. The idea of a payment channel. Suppose two parties, A and B, know that they will need to make a lot of payments to each other in the future. Instead of committing each payment as a transaction in the blockchain, they create a shared "money pool" (or perhaps a small private bank with exactly two accounts), and contribute some funds to it: A contributes a coins, and B contributes b coins. This is achieved by creating a special smart contract in the blockchain, and sending the money to it.

Before creating the "money pool", the two sides agree to a certain protocol. They will keep track of the *state* of the pool—that is, of their balances in the shared pool. Originally, the state is (a, b), meaning that a coins actually belong to A, and b coins belong to B. Then, if A wants to pay d coins to B, they can simply agree that the new state is $(a', b') = (a - d, b + d)$.

Afterwards, if, say, B wants to pay d' coins to A, the state will become $(a'', b'') = (a' + d', b' - d')$, and so on.

All this updating of balances inside the pool is done completely off-chain. When the two parties decide to withdraw their due funds from the pool, they do so according to the final state of the pool. This is achieved by sending a special message to the smart contract, containing the agreed-upon final state (a^*, b^*) along with the signatures of both A and B. Then the smart contract sends a^* coins to A, b^* coins to B and self-destructs.

This smart contract, along with the network protocol used by A and B to update the state of the pool, is a simple *payment channel between A and B*. According to the classification described in **5.1.2**, it is a *mixed* service: part of its state resides in the blockchain (the smart contract), but most of its state updates are performed off-chain (by the network protocol). If everything goes well, the two parties will be able to perform as many payments to each other as they want (with the only restriction being that the "capacity" of the channel is not overrun—i.e., their balances in the payment channel both remain non-negative), committing only two transactions into the blockchain: one to open (create) the payment channel (smart contract), and another to close (destroy) it.

6.1.2. Trustless payment channels. The previous example was somewhat unrealistic, because it assumes that both parties are willing to cooperate and will never cheat to gain some advantage. Imagine, for example, that A will choose not to sign the final balance (a', b') with $a' < a$. This would put B in a difficult situation.

To protect against such scenarios, one usually tries to develop *trustless* payment channel protocols, which do not require the parties to trust each other, and make provisions for punishing any party who would attempt to cheat.

This is usually achieved with the aid of signatures. The payment channel smart contract knows the public keys of A and B, and it can check their signatures if needed. The payment channel protocol requires the parties to sign the intermediate states and send the signatures to each other. Then, if one of the parties cheats—for instance, pretends that some state of the payment channel never existed—its misbehavior can be proved by showing its signature on that state. The payment channel smart contract acts as an "on-chain arbiter", able to process complaints of the two parties about each other, and punish the guilty party by confiscating all of its money and

awarding it to the other party.

6.1.3. Simple bidirectional synchronous trustless payment channel. Consider the following, more realistic example: Let the state of the payment channel be described by triple (δ_i, i, o_i), where i is the sequence number of the state (it is originally zero, and then it is increased by one when a subsequent state appears), δ_i is the *channel imbalance* (meaning that A and B own $a+\delta_i$ and $b-\delta_i$ coins, respectively), and o_i is the party allowed to generate the next state (either A or B). Each state must be signed both by A and B before any further progress can be made.

Now, if A wants to transfer d coins to B inside the payment channel, and the current state is $S_i = (\delta_i, i, o_i)$ with $o_i = A$, then it simply creates a new state $S_{i+1} = (\delta_i - d, i+1, o_{i+1})$, signs it, and sends it to B along with its signature. Then B confirms it by signing and sending a copy of its signature to A. After that, both parties have a copy of the new state with both of their signatures, and a new transfer may occur.

If A wants to transfer coins to B in a state S_i with $o_i = B$, then it first asks B to commit a subsequent state S_{i+1} with the same imbalance $\delta_{i+1} = \delta_i$, but with $o_{i+1} = A$. After that, A will be able to make its transfer.

When the two parties agree to close the payment channel, they both put their special *final* signatures on the state S_k they believe to be final, and invoke the *clean* or *two-sided finalization method* of the payment channel smart contract by sending it the final state along with both final signatures.

If the other party does not agree to provide its final signature, or simply if it stops responding, it is possible to close the channel unilaterally. For this, the party wishing to do so will invoke the *unilateral finalization* method, sending to the smart contract its version of the final state, its final signature, and the most recent state having a signature of the other party. After that, the smart contract does not immediately act on the final state received. Instead, it waits for a certain period of time (e.g., one day) for the other party to present its version of the final state. When the other party submits its version and it turns out to be compatible with the already submitted version, the "true" final state is computed by the smart contract and used to distribute the money accordingly. If the other party fails to present its version of the final state to the smart contract, then the money is redistributed according to the only copy of the final state presented.

If one of the two parties cheats—for example, by signing two different states as final, or by signing two different next states S_{i+1} and S'_{i+1}, or by

signing an invalid new state S_{i+1} (e.g., with imbalance $\delta_{i+1} < -a$ or $> b$)—then the other party may submit proof of this misbehavior to a third method of the smart contract. The guilty party is punished immediately by losing its share in the payment channel completely.

This simple payment channel protocol is *fair* in the sense that any party can always get its due, with or without the cooperation of the other party, and is likely to lose all of its funds committed to the payment channel if it tries to cheat.

6.1.4. Synchronous payment channel as a simple virtual blockchain with two validators. The above example of a simple synchronous payment channel can be recast as follows. Imagine that the sequence of states S_0, S_1, ..., S_n is actually the sequence of blocks of a very simple blockchain. Each block of this blockchain contains essentially only the current state of the blockchain, and maybe a reference to the previous block (i.e., its hash). Both parties A and B act as validators for this blockchain, so every block must collect both of their signatures. The state S_i of the blockchain defines the designated producer o_i for the next block, so there is no race between A and B for producing the next block. Producer A is allowed to create blocks that transfer funds from A to B (i.e., decrease the imbalance: $\delta_{i+1} \leq \delta_i$), and B can only transfer funds from B to A (i.e., increase δ).

If the two validators agree on the final block (and the final state) of the blockchain, it is finalized by collecting special "final" signatures of the two parties, and submitting them along with the final block to the channel smart contract for processing and re-distributing the money accordingly.

If a validator signs an invalid block, or creates a fork, or signs two different final blocks, it can be punished by presenting a proof of its misbehavior to the smart contract, which acts as an "on-chain arbiter" for the two validators; then the offending party will lose all its money kept in the payment channel, which is analogous to a validator losing its stake.

6.1.5. Asynchronous payment channel as a virtual blockchain with two workchains. The synchronous payment channel discussed in **6.1.3** has a certain disadvantage: one cannot begin the next transaction (money transfer inside the payment channel) before the previous one is confirmed by the other party. This can be fixed by replacing the single virtual blockchain discussed in **6.1.4** by a system of two interacting virtual workchains (or rather shardchains).

The first of these workchains contains only transactions by A, and its blocks can be generated only by A; its states are $S_i = (i, \phi_i, j, \psi_j)$, where i is the block sequence number (i.e., the count of transactions, or money transfers, performed by A so far), ϕ_i is the total amount transferred from A to B so far, j is the sequence number of the most recent valid block in B's blockchain that A is aware of, and ψ_j is the amount of money transferred from B to A in its j transactions. A signature of B put onto its j-th block should also be a part of this state. Hashes of the previous block of this workchain and of the j-th block of the other workchain may be also included. Validity conditions for S_i include $\phi_i \geq 0$, $\phi_i \geq \phi_{i-1}$ if $i > 0$, $\psi_j \geq 0$, and $-a \leq \psi_j - \phi_i \leq b$.

Similarly, the second workchain contains only transactions by B, and its blocks are generated only by B; its states are $T_j = (j, \psi_j, i, \phi_i)$, with similar validity conditions.

Now, if A wants to transfer some money to B, it simply creates a new block in its workchain, signs it, and sends to B, without waiting for confirmation.

The payment channel is finalized by A signing (its version of) the final state of its blockchain (with its special "final signature"), B signing the final state of its blockchain, and presenting these two final states to the clean finalization method of the payment channel smart contract. Unilateral finalization is also possible, but in that case the smart contract will have to wait for the other party to present its version of the final state, at least for some grace period.

6.1.6. Unidirectional payment channels. If only A needs to make payments to B (e.g., B is a service provider, and A its client), then a unilateral payment channel can be created. Essentially, it is just the first workchain described in **6.1.5** without the second one. Conversely, one can say that the asynchronous payment channel described in **6.1.5** consists of two unidirectional payment channels, or "half-channels", managed by the same smart contract.

6.1.7. More sophisticated payment channels. Promises. We will see later in **6.2.4** that the "lightning network" (cf. **6.2**), which enables instant money transfers through chains of several payment channels, requires higher degrees of sophistication from the payment channels involved.

In particular, we want to be able to commit "promises", or "conditional money transfers": A agrees to send c coins to B, but B will get the money

only if a certain condition is fulfilled, for instance, if B can present some string u with $\textsc{Hash}(u) = v$ for a known value of v. Otherwise, A can get the money back after a certain period of time.

Such a promise could easily be implemented on-chain by a simple smart contract. However, we want promises and other kinds of conditional money transfers to be possible off-chain, in the payment channel, because they considerably simplify money transfers along a chain of payment channels existing in the "lightning network" (cf. **6.2.4**).

The "payment channel as a simple blockchain" picture outlined in **6.1.4** and **6.1.5** becomes convenient here. Now we consider a more complicated virtual blockchain, the state of which contains a set of such unfulfilled "promises", and the amount of funds locked in such promises. This blockchain—or the two workchains in the asynchronous case—will have to refer explicitly to the previous blocks by their hashes. Nevertheless, the general mechanism remains the same.

6.1.8. Challenges for the sophisticated payment channel smart contracts. Notice that, while the final state of a sophisticated payment channel is still small, and the "clean" finalization is simple (if the two sides have agreed on their amounts due, and both have signed their agreement, nothing else remains to be done), the unilateral finalization method and the method for punishing fraudulent behavior need to be more complex. Indeed, they must be able to accept Merkle proofs of misbehavior, and to check whether the more sophisticated transactions of the payment channel blockchain have been processed correctly.

In other words, the payment channel smart contract must be able to work with Merkle proofs, to check their "hash validity", and must contain an implementation of *ev_trans* and *ev_block* functions (cf. **2.2.6**) for the payment channel (virtual) blockchain.

6.1.9. TON VM support for "smart" payment channels. The TON VM, used to run the code of TON Blockchain smart contracts, is up to the challenge of executing the smart contracts required for "smart", or sophisticated, payment channels (cf. **6.1.8**).

At this point the "everything is a bag of cells" paradigm (cf. **2.5.14**) becomes extremely convenient. Since all blocks (including the blocks of the ephemeral payment channel blockchain) are represented as bags of cells (and described by some algebraic data types), and the same holds for messages and Merkle proofs as well, a Merkle proof can easily be embedded into an

inbound message sent to the payment channel smart contract. The "hash condition" of the Merkle proof will be checked automatically, and when the smart contract accesses the "Merkle proof" presented, it will work with it as if it were a value of the corresponding algebraic data type—albeit incomplete, with some subtrees of the tree replaced by special nodes containing the Merkle hash of the omitted subtree. Then the smart contract will work with that value, which might represent, for instance, a block of the payment channel (virtual) blockchain along with its state, and will evaluate the *ev_block* function (cf. **2.2.6**) of that blockchain on this block and the previous state. Then either the computation finishes, and the final state can be compared with that asserted in the block, or an "absent node" exception is thrown while attempting to access an absent subtree, indicating that the Merkle proof is invalid.

In this way, the implementation of the verification code for smart payment channel blockchains turns out to be quite straightforward using TON Blockchain smart contracts. One might say that *the TON Virtual Machine comes with built-in support for checking the validity of other simple blockchains.* The only limiting factor is the size of the Merkle proof to be incorporated into the inbound message to the smart contract (i.e., into the transaction).

6.1.10. Simple payment channel within a smart payment channel. We would like to discuss the possibility of creating a simple (synchronous or asynchronous) payment channel inside an existing payment channel.

While this may seem somewhat convoluted, it is not much harder to understand and implement than the "promises" discussed in **6.1.7**. Essentially, instead of promising to pay c coins to the other party if a solution to some hash problem is presented, A promises to pay up to c coins to B according to the final settlement of some other (virtual) payment channel blockchain. Generally speaking, this other payment channel blockchain need not even be between A and B; it might involve some other parties, say, C and D, willing to commit c and d coins into their simple payment channel, respectively. (This possibility is exploited later in **6.2.5**.)

If the encompassing payment channel is asymmetric, two promises need to be committed into the two workchains: A will promise to pay $-\delta$ coins to B if the final settlement of the "internal" simple payment channel yields a negative final imbalance δ with $0 \leq -\delta \leq c$; and B will have to promise to pay δ to A if δ is positive. On the other hand, if the encompassing

payment channel is symmetric, this can be done by committing a single "simple payment channel creation" transaction with parameters (c, d) into the single payment channel blockchain by A (which would freeze c coins belonging to A), and then committing a special "confirmation transaction" by B (which would freeze d coins of B).

We expect the internal payment channel to be extremely simple (e.g., the simple synchronous payment channel discussed in **6.1.3**), to minimize the size of Merkle proofs to be submitted. The external payment channel will have to be "smart" in the sense described in **6.1.7**.

6.2 Payment Channel Network, or "Lightning Network"

Now we are ready to discuss the "lightning network" of TON Payments that enables instant money transfers between any two participating nodes.

6.2.1. Limitations of payment channels. A payment channel is useful for parties who expect a lot of money transfers between them. However, if one needs to transfer money only once or twice to a particular recipient, creating a payment channel with her would be impractical. Among other things, this would imply freezing a significant amount of money in the payment channel, and would require at least two blockchain transactions anyway.

6.2.2. Payment channel networks, or "lightning networks". Payment channel networks overcome the limitations of payment channels by enabling money transfers along *chains* of payment channels. If A wants to transfer money to E, she does not need to establish a payment channel with E. It would be sufficient to have a chain of payment channels linking A with E through several intermediate nodes—say, four payment channels: from A to B, from B to C, from C to D and from D to E.

6.2.3. Overview of payment channel networks. Recall that a *payment channel network*, known also as a "lightning network", consists of a collection of participating nodes, some of which have established long-lived payment channels between them. We will see in a moment that these payment channels will have to be "smart" in the sense of **6.1.7**. When a participating node A wants to transfer money to any other participating node E, she tries to find a path linking A to E inside the payment channel network. When such a path is found, she performs a "chain money transfer" along this path.

6.2.4. Chain money transfers. Suppose that there is a chain of payment channels from A to B, from B to C, from C to D, and from D to E. Suppose, further, that A wants to transfer x coins to E.

A simplistic approach would be to transfer x coins to B along the existing payment channel, and ask him to forward the money further to C. However, it is not evident why B would not simply take the money for himself. Therefore, one must employ a more sophisticated approach, not requiring all parties involved to trust each other.

This can be achieved as follows. A generates a large random number u and computes its hash $v = \textsc{Hash}(u)$. Then she creates a promise to pay x coins to B if a number u with hash v is presented (cf. **6.1.7**), inside her payment channel with B. This promise contains v, but not u, which is still kept secret.

After that, B creates a similar promise to C in their payment channel. He is not afraid to give such a promise, because he is aware of the existence of a similar promise given to him by A. If C ever presents a solution of the hash problem to collect x coins promised by B, then B will immediately submit this solution to A to collect x coins from A.

Then similar promises of C to D and of D to E are created. When the promises are all in place, A triggers the transfer by communicating the solution u to all parties involved—or just to E.

Some minor details are omitted in this description. For example, these promises must have different expiration times, and the amount promised might slightly differ along the chain (B might promise only $x - \epsilon$ coins to C, where ϵ is a small pre-agreed transit fee). We ignore such details for the time being, because they are not too relevant for understanding how payment channels work and how they can be implemented in TON.

6.2.5. Virtual payment channels inside a chain of payment channels. Now suppose that A and E expect to make a lot of payments to each other. They might create a new payment channel between them in the blockchain, but this would still be quite expensive, because some funds would be locked in this payment channel. Another option would be to use chain money transfers described in **6.2.4** for each payment. However, this would involve a lot of network activity and a lot of transactions in the virtual blockchains of all payment channels involved.

An alternative is to create a virtual payment channel inside the chain linking A to E in the payment channel network. For this, A and E create

a (virtual) blockchain for their payments, as if they were going to create a payment channel in the blockchain. However, instead of creating a payment channel smart contract in the blockchain, they ask all intermediate payment channels—those linking A to B, B to C, etc.—to create simple payment channels inside them, bound to the virtual blockchain created by A and E (cf. **6.1.10**). In other words, now a promise to transfer money according to the final settlement between A and E exists inside every intermediate payment channel.

If the virtual payment channel is unidirectional, such promises can be implemented quite easily, because the final imbalance δ is going to be non-positive, so simple payment channels can be created inside intermediate payment channels in the same order as described in **6.2.4**. Their expiration times can also be set in the same way.

If the virtual payment channel is bidirectional, the situation is slightly more complicated. In that case, one should split the promise to transfer δ coins according to the final settlement into two half-promises, as explained in **6.1.10**: to transfer $\delta^- = \max(0, -\delta)$ coins in the forward direction, and to transfer $\delta^+ = \max(0, \delta)$ in the backward direction. These half-promises can be created in the intermediate payment channels independently, one chain of half-promises in the direction from A to E, and the other chain in the opposite direction.

6.2.6. Finding paths in the lightning network. One point remains undiscussed so far: how will A and E find a path connecting them in the payment network? If the payment network is not too large, an OSPF-like protocol can be used: all nodes of the payment network create an overlay network (cf. **4.2.17**), and then every node propagates all available link (i.e., participating payment channel) information to its neighbors by a gossip protocol. Ultimately, all nodes will have a complete list of all payment channels participating in the payment network, and will be able to find the shortest paths by themselves—for example, by applying a version of Dijkstra's algorithm modified to take into account the "capacities" of the payment channels involved (i.e., the maximal amounts that can be transferred along them). Once a candidate path is found, it can be probed by a special ADNL datagram containing the full path, and asking each intermediate node to confirm the existence of the payment channel in question, and to forward this datagram further according to the path. After that, a chain can be constructed, and a protocol for chain transfers (cf. **6.2.4**), or for creating a

virtual payment channel inside a chain of payment channels (cf. **6.2.5**), can be run.

6.2.7. Optimizations. Some optimizations might be done here. For example, only transit nodes of the lightning network need to participate in the OSPF-like protocol discussed in **6.2.6**. Two "leaf" nodes wishing to connect through the lightning network would communicate to each other the lists of transit nodes they are connected to (i.e., with which they have established payment channels participating in the payment network). Then paths connecting transit nodes from one list to transit nodes from the other list can be inspected as outlined above in **6.2.6**.

6.2.8. Conclusion. We have outlined how the blockchain and network technologies of the TON project are adequate to the task of creating *TON Payments*, a platform for off-chain instant money transfers and micropayments. This platform can be extremely useful for services residing in the TON ecosystem, allowing them to easily collect micropayments when and where required.

Conclusion

We have proposed a scalable multi-blockchain architecture capable of supporting a massively popular cryptocurrency and decentralized applications with user-friendly interfaces.

To achieve the necessary scalability, we proposed the *TON Blockchain*, a "tightly-coupled" multi-blockchain system (cf. **2.8.14**) with bottom-up approach to sharding (cf. **2.8.12** and **2.1.2**). To further increase potential performance, we introduced the 2-blockchain mechanism for replacing invalid blocks (cf. **2.1.17**) and Instant Hypercube Routing for faster communication between shards (cf. **2.4.20**). A brief comparison of the TON Blockchain to existing and proposed blockchain projects (cf. **2.8** and **2.9**) highlights the benefits of this approach for systems that seek to handle millions of transactions per second.

The *TON Network*, described in Chapter **4**, covers the networking demands of the proposed multi-blockchain infrastructure. This network component may also be used in combination with the blockchain to create a wide spectrum of applications and services, impossible using the blockchain alone (cf. **2.9.13**). These services, discussed in Chapter **5**, include *TON DNS*, a service for translating human-readable object identifiers into their addresses; *TON Storage*, a distributed platform for storing arbitrary files; *TON Proxy*, a service for anonymizing network access and accessing TON-powered services; and *TON Payments* (cf. Chapter **6**), a platform for instant off-chain money transfers across the TON ecosystem that applications may use for micropayments.

The TON infrastructure allows for specialized light client wallet and "ton-browser" desktop and smartphone applications that enable a browser-like experience for the end user (cf. **5.3.24**), making cryptocurrency payments and interaction with smart contracts and other services on the TON Platform accessible to the mass user. Such a light client can be integrated into the Telegram Messenger client (cf. **5.3.19**), thus eventually bringing a wealth of blockchain-based applications to hundreds of millions of users.

References

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

[2] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[3] M. Ben-Or, B. Kelmer, T. Rabin, *Asynchronous secure computations with optimal resilience*, in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, p. 183–192. ACM, 1994.

[4] 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`.

[5] EOS.IO, *EOS.IO technical white paper*, `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`, 2017.

[6] D. Goldschlag, M. Reed, P. Syverson, *Onion Routing for Anonymous and Private Internet Connections*, *Communications of the ACM*, **42**, num. 2 (1999), `http://www.onion-router.net/Publications/CACM-1999.pdf`.

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

[8] S. Larimer, *The history of BitShares*, `https://docs.bitshares.org/bitshares/history.html`, 2013.

[9] M. Luby, A. Shokrollahi, et al., *RaptorQ forward error correction scheme for object delivery*, IETF RFC 6330, `https://tools.ietf.org/html/rfc6330`, 2011.

[10] P. Maymounkov, D. Mazières, *Kademlia: A peer-to-peer information system based on the XOR metric*, in *IPTPS '01 revised papers from the First International Workshop on Peer-to-Peer Systems*,

p. 53–65, available at http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf, 2002.

[11] 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.

[12] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, https://bitcoin.org/bitcoin.pdf, 2008.

[13] S. Peyton Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, *Journal of Functional Programming* **2** (2), p. 127–202, 1992.

[14] A. Shokrollahi, M. Luby, *Raptor Codes*, *IEEE Transactions on Information Theory* **6**, no. 3–4 (2006), p. 212–322.

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

[16] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013, available at https://homotopytypetheory.org/book.

[17] G. Wood, *PolkaDot: vision for a heterogeneous multi-chain framework*, draft 1, https://github.com/w3f/polkadot-white-paper/raw/master/PolkaDotPaper.pdf, 2016.

A The TON Coin, or the Gram

The principal cryptocurrency of the TON Blockchain, and in particular of its masterchain and basic workchain, is the *TON Coin*, also known as the *Gram* (GRM). It is used to make deposits required to become a validator; transaction fees, gas payments (i.e., smart-contract message processing fees) and persistent storage payments are also usually collected in Grams.

A.1. Subdivision and terminology. A *Gram* is subdivided into one billion (10^9) smaller units, called *nanograms*, *ngrams* or simply *nanos*. All transfers and account balances are expressed as non-negative integer multiples of nanos. Other units include:

- A *nano*, *ngram* or *nanogram* is the smallest unit, equal to 10^{-9} Grams.
- A *micro* or *microgram* equals one thousand (10^3) nanos.
- A *milli* is one million (10^6) nanos, or one thousandth part (10^{-3}) of a Gram.
- A *Gram* equals one billion (10^9) nanos.
- A *kilogram*, or *kGram*, equals one thousand (10^3) Grams.
- A *megagram*, or *MGram*, equals one million (10^6) Grams, or 10^{15} nanos.
- Finally, a *gigagram*, or *GGram*, equals one billion (10^9) Grams, or 10^{18} nanos.

There will be no need for larger units, because the initial supply of Grams will be limited to five billion ($5 \cdot 10^9$) Grams (i.e., 5 Gigagrams).

A.2. Smaller units for expressing gas prices. If the necessity for smaller units arises, "specks" equal to 2^{-16} nanograms will be used. For example, gas prices may be indicated in specks. However, the actual fee to be paid, computed as the product of the gas price and the amount of gas consumed, will be always rounded down to the nearest multiple of 2^{16} specks and expressed as an integer number of nanos.

A.3. Original supply, mining rewards and inflation. The total supply of Grams is originally limited to 5 Gigagrams (i.e., five billion Grams or $5 \cdot 10^{18}$ nanos).

This supply will increase very slowly, as rewards to validators for mining new masterchain and shardchain blocks accumulate. These rewards would amount to approximately 20% (the exact number may be adjusted in future) of the validator's stake per year, provided the validator diligently performs its duties, signs all blocks, never goes offline and never signs invalid blocks. In this way, the validators will have enough profit to invest into better and faster hardware needed to process the ever growing quantity of users' transactions.

We expect that at most 10%[33] of the total supply of Grams, on average, will be bound in validator stakes at any given moment. This will produce an inflation rate of 2% per year, and as a result, will double the total supply of Grams (to ten Gigagrams) in 35 years. Essentially, this inflation represents a payment made by all members of the community to the validators for keeping the system up and running.

On the other hand, if a validator is caught misbehaving, a part or all of its stake will be taken away as a punishment, and a larger portion of it will subsequently be "burned", decreasing the total supply of Grams. This would lead to deflation. A smaller portion of the fine may be redistributed to the validator or the "fisherman" who committed a proof of the guilty validator's misbehavior.

A.4. Original price of Grams. The price of the first Gram to be sold will equal approximately \$0.1 (USD). Every subsequent Gram to be sold (by the TON Reserve, controlled by the TON Foundation) will be priced one billionth higher than the previous one. In this way, the n-th Gram to be put into circulation will be sold at approximately

$$p(n) \approx 0.1 \cdot (1 + 10^{-9})^n \quad \text{USD}, \tag{26}$$

or an approximately equivalent (because of quickly changing market exchange rates) amount of other (crypto)currencies, such as BTC or ETH.

A.4.1. Exponentially priced cryptocurrencies. We say that the Gram is an *exponentially priced cryptocurrency*, meaning that the price of the n-th

[33]The maximum total amount of validator stakes is a configurable parameter of the blockchain, so this restriction can be enforced by the protocol if necessary.

Gram to be put into circulation is approximately $p(n)$ given by the formula

$$p(n) = p_0 \cdot e^{\alpha n} \tag{27}$$

with specific values $p_0 = 0.1$ USD and $\alpha = 10^{-9}$.

More precisely, a small fraction dn of a new coin is worth $p(n)\,dn$ dollars, once n coins are put into circulation. (Here n is not necessarily an integer.)

Other important parameters of such a cryptocurrency include n, the total number of coins in circulation, and $N \geq n$, the total number of coins that can exist. For the Gram, $N = 5 \cdot 10^9$.

A.4.2. Total price of first n coins. The total price $T(n) = \int_0^n p(n)\,dn \approx p(0) + p(1) + \cdots + p(n-1)$ of the first n coins of an exponentially priced cryptocurrency (e.g., the Gram) to be put into circulation can be computed by

$$T(n) = p_0 \cdot \alpha^{-1}(e^{\alpha n} - 1) \quad . \tag{28}$$

A.4.3. Total price of next Δn coins. The total price $T(n + \Delta n) - T(n)$ of Δn coins put into circulation after n previously existing coins can be computed by

$$T(n+\Delta n) - T(n) = p_0 \cdot \alpha^{-1}(e^{\alpha(n+\Delta n)} - e^{\alpha n}) = p(n) \cdot \alpha^{-1}(e^{\alpha\,\Delta n} - 1) \quad . \tag{29}$$

A.4.4. Buying next coins with total value T. Suppose that n coins have already been put into circulation, and that one wants to spend T (dollars) on buying new coins. The quantity of newly-obtained coins Δn can be computed by putting $T(n + \Delta n) - T(n) = T$ into (29), yielding

$$\Delta n = \alpha^{-1} \log\left(1 + \frac{T \cdot \alpha}{p(n)}\right) \quad . \tag{30}$$

Of course, if $T \lll p(n)\alpha^{-1}$, then $\Delta n \approx T/p(n)$.

A.4.5. Market price of Grams. Of course, if the free market price falls below $p(n) := 0.1 \cdot (1+10^{-9})^n$, once n Grams are put into circulation, nobody would buy new Grams from the TON Reserve; they would choose to buy their Grams on the free market instead, without increasing the total quantity of Grams in circulation. On the other hand, the market price of a Gram cannot become much higher than $p(n)$, otherwise it would make sense to obtain new Grams from the TON Reserve. This means that the market price of Grams

would not be subject to sudden spikes (and drops); this is important because stakes (validator deposits) are frozen for at least one month, and gas prices cannot change too fast either. So, the overall economic stability of the system requires some mechanism that would prevent the exchange rate of the Gram from changing too drastically, such as the one described above.

A.4.6. Buying back the Grams. If the market price of the Gram falls below $0.5 \cdot p(n)$, when there are a total of n Grams in circulation (i.e., not kept on a special account controlled by the TON Reserve), the TON Reserve reserves the right to buy some Grams back and decrease n, the total quantity of Grams in circulation. This may be required to prevent sudden falls of the Gram exchange rate.

A.4.7. Selling new Grams at a higher price. The TON Reserve will sell only up to one half (i.e., $2.5 \cdot 10^9$ Grams) of the total supply of Grams according to the price formula (26). It reserves the right not to sell any of the remaining Grams at all, or to sell them at a higher price than $p(n)$, but never at a lower price (taking into account the uncertainty of quickly changing exchange rates). The rationale here is that once at least half of all Grams have been sold, the total value of the Gram market will be sufficiently high, and it will be more difficult for outside forces to manipulate the exchange rate than it may be at the very beginning of the Gram's deployment.

A.5. Using unallocated Grams. The TON Reserve will use the bulk of "unallocated" Grams (approximately $5 \cdot 10^9 - n$ Grams)—i.e., those residing in the special account of the TON Reserve and some other accounts explicitly linked to it—only as validator stakes (because the TON Foundation itself will likely have to provide most of the validators during the first deployment phase of the TON Blockchain), and for voting in the masterchain for or against proposals concerning changes in the "configurable parameters" and other protocol changes, in the way determined by the TON Foundation (i.e., its creators—the development team). This also means that the TON Foundation will have a majority of votes during the first deployment phase of the TON Blockchain, which may be useful if a lot of parameters end up needing to be adjusted, or if the need arises for hard or soft forks. Later, when less than half of all Grams remain under control of the TON Foundation, the system will become more democratic. Hopefully it will have become more mature by then, without the need to adjust parameters too frequently.

A.5.1. Some unallocated Grams will be given to developers. A predefined (relatively small) quantity of "unallocated" Grams (e.g., 200 Megagrams, equal to 4% of the total supply) will be transferred during the deployment of the TON Blockchain to a special account controlled by the TON Foundation, and then some "rewards" may be paid from this account to the developers of the open source TON software, with a minimum two-year vesting period.

A.5.2. The TON Foundation needs Grams for operational purposes. Recall that the TON Foundation will receive the fiat and cryptocurrency obtained by selling Grams from the TON Reserve, and will use them for the development and deployment of the TON Project. For instance, the original set of validators, as well as an initial set of TON Storage and TON Proxy nodes may be installed by the TON Foundation.

While this is necessary for the quick start of the project, the ultimate goal is to make the project as decentralized as possible. To this end, the TON Foundation may need to encourage installation of third-party validators and TON Storage and TON Proxy nodes—for example, by paying them for storing old blocks of the TON Blockchain or proxying network traffic of a selected subset of services. Such payments will be made in Grams; therefore, the TON Foundation will need a significant amount of Grams for operational purposes.

A.5.3. Taking a pre-arranged amount from the Reserve. The TON Foundation will transfer to its account a small part of the TON Reserve—say, 10% of all coins (i.e. 500 Megagrams) after the end of the initial sale of Grams—to be used for its own purposes as outlined in **A.5.2**. This is best done simultaneously with the transfer of the funds intended for TON developers, as mentioned in **A.5.1**.

After the transfers to the TON Foundation and the TON developers, the TON Reserve price $p(n)$ of the Gram will immediately rise by a certain amount, known in advance. For example, if 10% of all coins are transferred for the purposes of the TON Foundation, and 4% are transferred for the encouragement of the developers, then the total quantity n of coins in circulation will immediately increase by $\Delta n = 7 \cdot 10^8$, with the price of the Gram multiplying by $e^{\alpha\,\Delta n} = e^{0.7} \approx 2$ (i.e, doubling).

The remainding "unallocated" Grams will be used by the TON Reserve as explained above in **A.5**. If the TON Foundation needs any more Grams

thereafter, it will simply convert into Grams some of the funds it had previously obtained during the sale of the coins, either on the free market or by buying Grams from the TON Reserve. To prevent excessive centralization, the TON Foundation will never endeavour to have more than 10% of the total amount of Grams (i.e., 500 Megagrams) on its account.

A.6. Bulk sales of Grams. When a lot of people simultaneously want to buy large amounts of Grams from the TON Reserve, it makes sense not to process their orders immediately, because this would lead to results very dependent on the timing of specific orders and their processing sequence.

Instead, orders for buying Grams may be collected during some predefined period of time (e.g., a day or a month) and then processed all together at once. If k orders with i-th order worth T_i dollars arrive, then the total amount $T = T_1 + T_2 + \cdots + T_k$ is used to buy Δn new coins according to (30), and the sender of the i-th order is allotted $\Delta n \cdot T_i/T$ of these coins. In this way, all buyers obtain their Grams at the same average price of $T/\Delta n$ USD per Gram.

After that, a new round of collecting orders for buying new Grams begins.

When the total value of Gram buying orders becomes low enough, this system of "bulk sales" may be replaced with a system of immediate sales of Grams from the TON Reserve according to formula (30).

The "bulk sales" mechanism will probably be used extensively during the initial phase of collecting investments in the TON Project.