Hydra: Support Dynamic BFT With Weaker Assumptions and Explicit Request Handling

Xuyang Liu, *Graduate Student Member, IEEE*, Zijian Zhang, *Senior Member, IEEE*, Zhen Li, Haibo Sun, *Graduate Student Member, IEEE*, Meng Li, *Senior Member, IEEE*, Jing Sun, Jiamou Liu, Yong Liu, Lei Xu, Jincheng An, Qi Sun, Liang Huang, Mauro Conti, *Fellow, IEEE*, and Liehuang Zhu, *Senior Member, IEEE*

***Abstract*—This paper presents Hydra, a dynamic BFT protocol that allows replicas to join and leave the system dynamically. It addresses the limitations of traditional static BFTs in managing membership changes and can be used to simplify the implementation of many features in modern blockchain applications. Hydra relies on weaker assumptions to achieve standard properties compared to the existing solution Dyno and introduces a configuration auto-transition protocol to ensure liveness. Through temporary configurations and explicitly defined replica responsibilities for request handling, Hydra pipelines membership requests alongside regular requests and realizes clarity, achieving a more efficient and smoother configuration transitions. It also employs a non-blocking configuration discovery mechanism, enabling new replicas to participate in consensus quickly. We formally prove Hydra's correctness under the dynamic BFT model. Experimental results demonstrate Hydra's ability to maintain throughput fluctuations within 5% during various replica join and leave scenarios, outperforming Dyno and existing BFT system supporting reconfiguration in both stability and efficiency. Hydra effectively manages scenarios that Dyno circumvents with stronger assumptions and quickly restores throughput to normal levels.**

***Index Terms*—Blockchain, consensus protocol, Byzantine fault tolerance, dynamic membership service.**

Received 23 November 2024; revised 1 December 2025; accepted 16 December 2025. Date of publication 22 December 2025; date of current version 12 May 2026. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant U2468205, Grant U23A20303, Grant 62372149, Grant 62572168, and Grant 625B2022, and in part by Anhui Provincial Natural Science Foundation under Grant 2508085MF151, and in part by the Key Laboratory of Knowledge Engineering with Big Data (the Ministry of Education of China), under Grant BigKEOpen2025-04, and in part by the Open Foundation of State key Laboratory of Networking and Switching Technology (Beijing University of Posts and Telecommunications) under Grant SKLNST-2025-1-12, and in part by EU LOCARD Project under Grant H2020-SU-SEC-2018-832735. *(Corresponding authors: Zijian Zhang; Meng Li.)*

Xuyang Liu is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, and also with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: liuxuyang@bit.edu.cn).

Zijian Zhang, Zhen Li, Haibo Sun, Lei Xu, and Liehuang Zhu are with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: zhangzijian@bit.edu.cn; zhen.li@bit.edu.cn; sunhypper@bit.edu.cn; xu.lei@bit.edu.cn; liehuangz@bit.edu.cn).

Meng Li is with the School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230002, China, and with the State key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China, and also with the Key Laboratory of Knowledge Engineering with Big Data, Ministry of Education, Hefei University of Technology, Hefei 230601, China (e-mail: mengli@hfut.edu.cn).

Jing Sun and Jiamou Liu are with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: jing.sun@auckland.ac.nz; jiamou.liu@auckland.ac.nz).

Yong Liu, Jincheng An, and Liang Huang are with the Qianxin Technology Group Company, Ltd, Beijing 100088, China (e-mail: liuyong03@qianxin.com; anjincheng@qianxin.com; huangliang@qianxin.com).

Qi Sun is with the Hangzhou Nuowei Information Technology Company Ltd., Hangzhou 310000, China (e-mail: sunq0810@gmail.com).

Mauro Conti is with the Department of Mathematics and HIT Center, University of Padua, 35121 Padua, Italy, and also with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2600 Delft, Netherlands (e-mail: mauro.conti@unipd.it).

This article has supplementary downloadable material available at https://doi.org/10.1109/TDSC.2025.3647269, provided by the authors.

Digital Object Identifier 10.1109/TDSC.2025.3647269

I. Introduction

BYZANTINE Fault Tolerance (BFT) consensus mechanism is designed to ensure system reliability and consistency despite failures. Initially conceptualized through the Byzantine Generals' Problem [1], BFT was applied in distributed systems, fault-tolerant computing, and networking. The first significant application of BFT in blockchain technology was by Tendermint [2] to enhance the efficiency and security of blockchain consensus, addressing limitations of earlier mechanisms like Proof of Work (PoW) [3].

In permissioned blockchains [4], where participant identities are known and controlled, BFT protocols offer robust security and efficiency, enabling rapid consensus and stronger fault tolerance. However, traditional BFT protocols are restrictive due to their static nature, lacking support for dynamic membership changes. Conversely, permissionless blockchains [5] face greater challenges due to their open, decentralized nature, where participants can join and leave freely. This dynamic environment complicates the implementation of BFT, requiring sophisticated mechanisms to maintain consensus amidst an ever-changing set of replicas. Common approaches include selecting a random subset of replicas through PoW [6] or Proof of Stake (PoS) [7] to form a committee for consensus. Some permissionless blockchains, like Cosmos [8], also use hybrid consensus models for finalizing.

This paper studies the integration of dynamic membership services into BFT models, whereby replicas may join and leave the system dynamically. Dynamic BFT represents a viable alternative approach for permissionless blockchains and a support for dynamic-committee-based BFT-friendly protocols [9], [10].

It can also be employed to facilitate reconfiguration in various scenarios, such as long-running consortium blockchains where participant changes are inevitable [11]. Dynamic BFT-based reconfiguration manages these changes without disrupting the system. Moreover, dynamic BFT provides potential approaches to passive [12] and proactive recovery [13] by allowing fresh or new replicas to replace the faulty ones when recovery is infeasible or to periodically replace the potentially malicious ones when adversaries' perpetuation is a concern, enhancing long-term security.

Technical issues and Our Contributions: Dynamic group communication [14] in distributed systems has been extensively studied, offering various primitives and valuable syntax for reliable communication and membership management in dynamic environments. Nevertheless, these schemes typically assume crash failures [12], [15]. While some researches addresses BFT [16], [17], they concentrate on the broadcast problem rather than consensus. To illustrate, DBRB [17] permits divergent view paths and ensures eventual convergence to the same view, which is not compatible with consensus from a technical perspective. While some BFT schemes [18], [19] also focusing on flexible addition and removal of replicas, they lack a good abstraction for dynamic membership service with provable security, especially in addressing the technical challenges specific to dynamic BFT. Most intuitively, membership changes alter quorum sizes, complicating consensus when replicas join or leave. Moreover, it is assumed that valid requests are eventually committed by all correct replicas in static systems [20], but the same becomes complicated in dynamic systems. Not all replicas in a given configuration are necessarily required to commit a proposal. It is also possible some correct replicas may leave the system before requests are committed, violating vital properties.

Dyno [21], as the pioneering solution, offers the first formal treatment of dynamic BFT. It provides a rigorous formalization of dynamic BFT, including a set of security definitions that encompass both regular and membership requests. Dyno designed configuration discovery protocols for clients and new replicas obtain the system membership, and constructed the normal state protocol using PBFT [22] notations. While Dyno achieves the dynamic membership service in BFT, it is evident that there are significant practical limitations:

- *Overly strong assumptions of membership correctness:* Dyno's standard assumption posits that the system maintains optimal resilience (i.e., correct replicas always form a quorum) across all configurations, without restricting who may issue membership requests, which is overly strong. For instance, in a configuration where correct replicas precisely form a quorum, the leaving of a correct replica or the joining of a malicious one can lead to a fault tolerance breach. Under Dyno's assumptions, it is critical to ensure that correct replicas leave or join in tandem with malicious replicas to maintain resilience. Additionally, under the standard assumptions, Dyno cannot achieve standard agreement property in committing requests and requires a stronger assumption (i.e., that more correct replicas than the maximum tolerable faults in any possible configuration always exist in the system) to do so.
- *Slightly chaotic request handling:* In Dyno, if a consensus round includes membership requests, votes from new members having initiated valid join requests (remain uncommitted) but not yet part of the current configuration are considered valid for forming a quorum, though the quorum size is still based on the current configuration. Additionally, replicas initiate leave requests have a fixed waiting period before leaving the system, which means correct replicas might leave before committing all their requests.
- *Blocking when performing configuration discovery:* In Dyno, replicas must first obtain the latest membership through configuration discovery before initiating requests, and verify its validity from the initial configuration using the complete configuration history it received together. Since the above process is blocking, if a configuration transition occurs while a replica is waiting in discovery, the replica may need to re-execute the protocol later. This can significantly impact the efficiency of configuration transitions, especially in systems where configurations are high or change frequently, potentially leading to a backlog of new replicas unable to join the system promptly.

The aforementioned limitations of Dyno reveal fundamental challenges in designing practical dynamic BFT systems. First, relaxing the overly strong membership correctness assumptions introduces scenarios where the number of faulty replicas may exceed the tolerable threshold (e.g., in a configuration where the number of faults is at the tolerable threshold, and then one correct replica leaves). Second, achieving clear request handling and standard properties require carefully defining replica responsibilities and voting periods. Third, new or lagging replicas must learn a continuous, verifiable configuration history without blocking normal consensus, otherwise frequent transitions can prevent timely participation.

We propose Hydra, a dynamic BFT paradigm with a construction path not quite the same as Dyno. The objective is to resolve the aforementioned issues, to achieve a better performance, and to be proven secure under the standard security definition of the dynamic BFT model. Specifically, our approach has the following characteristics:

- *Weaker assumptions on membership correctness:* Hydra requires that correct replicas just form a quorum among all members. Additionally, it requires more correct replicas than half of the maximum tolerable faults never leave the system, and they participate in leader rotation (a subset of these replicas are permitted to experience a temporary crash fault, provided they can subsequently be recovered). Beyond that, there are no requirements for the composition of any configuration. This introduces the possibility of configurations where malicious replicas exceeds fault tolerance. To address this, we introduce an configuration auto-transition protocol to ensure system liveness: after a timeout, the leader identifies replicas with inconsistent behavior and automatically initiates leave requests for them.
- *Temporary configuration and replica responsibility:* Hydra explicitly defines replica responsibilities: for any request, the membership of the configuration where the quorum votes is collected for the first time are responsible for

that request. Quorum proof is constructed only among responsible replicas, and non-responsible replicas do not need to confirm or commit the request. We also introduce the temporary configurations - an effective way to pipeline membership requests and achieve a more stable performance: replicas initiating join requests enter a temporary configuration and move to a formal one upon requests commitment. Replicas in the temporary but informal configuration are not responsible for any requests; their votes only validate the temporary configuration. Correct replicas initiating leave requests must complete all their responsible requests before leaving the system. By defining replicas responsibilities, Hydra introduces and achieves Clarity, a stronger property than Dyno's consistent delivery, clarifying which membership ultimately commits and responds.

- *Non-blocking configuration discovery:* In Hydra, new and offline replicas first skip non-contiguous parts of configurations and follow the highest valid configuration they know to continue the consensus protocol while gradually complete the missing historical configurations through a non-blocking configuration discovery. This ensures an efficient discovery and transition, allowing new replicas to quickly participate in consensus even when configurations are high or change frequently. Each correct replica maintains a continuous and verifiable configuration list locally.

We formally prove the correctness of Hydra under the dynamic BFT model. Experimental results show that Hydra maintains throughput fluctuations within 5% in various replica join and leave scenarios, achieving higher and more stable performance than both Dyno and the BFT system supporting reconfiguration when handling membership requests. Hydra effectively manages scenarios where faults join or existing correct replicas leave when faults reach the tolerance threshold, which Dyno circumvents with stronger assumptions, and quickly restores throughput to normal levels.

II. Related Work

BFT: In 1982, Lamport et al. introduced the Byzantine Generals Problem [1], framing the challenge of reaching consensus in the presence of faulty or malicious replicas. The initial solutions [27], [28], [29] to this problem laid the foundation for BFT, albeit with high complexity. Castro and Liskov introduced the Practical Byzantine Fault Tolerance (PBFT) [22] protocol in 1999, addressing the inefficiencies of earlier BFT models, making it feasible for real-world applications.

BFT was initially developed to address reliability and security issues in traditional distributed systems. However, its principles proved to be incredibly beneficial for the emerging blockchain technologies. Permissioned blockchains, like Hyperledger Fabric [30], leverage PBFT and its variants to get a group of identified, pre-selected validators to agree on a solution. Permissionless blockchains face challenges with BFT due to their dynamic and open nature. Solutions like Ethereum 2.0's Casper protocol [31] integrate PoS with BFT-style consensus for finality and security. Innovations include random selection of replicas to form consensus committees, as seen in Algorand [6] (PoS election using Verifiable Random Function), Rapidchain [7] (PoW election with separate committees in each shard) and GRBFT/S-GRBFT [24] (reputation and PoS-based hybrid committee selection with multilateral Gaussian evaluation). A number of blockchain-specific BFT protocols have also been proposed, exemplified by Tendermint [2] (block locking), Casper [31] (pipelining techniques), HotStuff [25] (streamlined chain structure and optimistically responsive) and their variants [26], [32], [33].

Dynamic group communication: Dynamic group communication addresses the need for systems to adapt to changing membership over time. Birman and Joseph [14] first introduced group membership abstractions that provide a dynamic yet consistent view of active members. They extended this with view synchronous communication [34], ensuring reliable message delivery within the current group members. The notion of extended virtual synchrony [35] further refines this by maintaining a consistent relationship between message delivery and configuration changes, ensuring that all replicas observe the same sequence of events relative to these changes.

Systems like Spread [36], Secure Spread [37] implement these principles in the crash failure model, with Secure Spread adding security layers such as authentication and integrity. Some implementations [16], [38] address Byzantine failures using Byzantine failure detectors for liveness. Chockler et al. [15] conducted an extensive survey of dynamic group communication systems, where the group membership service is defined first and serves as the basic layer of various communication stacks, followed by the specification of communication primitives. They also categorize membership services into two models: primary partition and partitionable. Views in the latter one are partially ordered, allowing the existence of multiple disjoint views simultaneously. While in the primary partition model, which we focus on, views at all replicas are totally ordered. Schiper [12] proposed an alternative approach where communication primitives are defined first, followed by membership changes, allowing all membership changes to come from explicit invocations of membership requests.

Dynamic membership service for SMR: Dynamic membership service for State Machine Replication (SMR) has been studied in various systems, mostly under crash failure settings. One approach is to manage membership changes as part of the system state, as exemplified by Paxos [39]. SMART [40] extends this by creating a new replica group running simultaneous Paxos instances until the state fully migrates. Lamport et al. also introduced a configuration master for SMR configuration management [41]. Raft [42] uses a two-phase joint consensus configuration for smooth transitions in primary/backup replication. Membership management services, like Zookeeper [43], facilitate replica management and reconfiguration [44], [45]. In BFT, Sousa et al. introduced BFT-SMaRt [23], in which reconfiguration requests can be initiated by system administrators through a view manager client, and are ordered with regular requests.

Unlike conventional reconfiguration approaches, Dynamic BFT is built as a self-configurable SMR. Duan and Zhang provided the first formal treatment of dynamic BFT, identified

TABLE I
COMPARISON BETWEEN HYDRA AND REPRESENTATIVE BFT CONSENSUS PROTOCOLS IN TERMS OF MEMBERSHIP MANAGEMENT

Protocol	Category	Membership Model	Granularity (Membership Change)	Membership Correctness assumption	Explicit Request Handling	Non-Blocking Configuration Discovery
Hydra (ours)	Dynamic BFT	Dynamic on-the-fly	Per-replica	$(f < \frac{n}{3})$ and $(> \frac{f-1}{2}$ permanent correct replicas)[$]	Yes	Yes
Dyno [16]	Dynamic BFT	Dynamic on-the-fly	Per-replica	$\forall c \geq 0, (f_c < \frac{n_c}{3})$ or $(\geq max(f_c)+1$ correct replicas across all configurations)[†]	No	No
BFT-SMaRt [7]	Reconfigurable BFT (admin)	Admin-triggered reconfig	Per-replica (via view manager)	$f < \frac{n}{3}$	Yes	N/A (state transfer)
GRBFT / S-GRBFT [45]	Hybrid/ committee BFT	Committee per-epoch	Per-epoch committee	$\leq \frac{1}{3} - \epsilon$ adversarial stakeholding fraction[*]	Yes	N/A
Algorand [19]	Hybrid/ committee BFT	Committee per-round	Per-round committee	$\leq \frac{1}{3}$ adversarial stakeholding fraction	Yes	N/A
PBFT [12]	Static BFT	Static (fixed validators)	N/A	$f < \frac{n}{3}$	Yes	N/A
HotStuff [47] / Fast-HotStuff [23]	Static BFT	Static (fixed validators)	N/A	$f < \frac{n}{3}$	Yes	N/A

[$]: f is the number of replicas corrupted by the adversary, and n is the total number of replicas.
[†]: c denotes a configuration index. n_c and f_c are the number of replicas and the number of corrupted replicas in configuration c, respectively.
[*]: $0 < \epsilon < \frac{1}{3}$ is the restriction imposed on the adversary (see the original literature for further details).

issues, lifted security definitions and designed Dyno [21]. Dyno is regarded as a method applicable in Dynamic Consensus Committee-Based Secret Sharing [9], [10]. Some research suggests using K-Nearest Neighbour for malicious replica classification with Dyno to facilitate the differentiation of requests [19]. However, Dyno still faces practical limitations that require addressing, which is the focus of this paper.

Table I compares Hydra with representative BFT consensus protocols along the criteria established in Sections I–II. Rather than enumerating all protocols cited in the related work, we deliberately select a small but representative subset that covers all the classes discussed. Among these, Dyno is the most closely related to Hydra. However, we emphasise the table focuses on BFT consensus/SMR protocols. Our specification is also closely related to dynamic group communication abstractions, in particular Schiper's work on uniform broadcast with dynamic membership. These works target a different layer (group communication and view management) and therefore do not fit directly into the BFT-consensus-oriented table. A more detailed qualitative comparison with the most closely related prior work, including Dyno and Schiper's dynamic membership service, is provided in Section III-D, where we further highlight Hydra's distinctive improvements and design objectives.

III. PROBLEM STATEMENT

A. System Model

A dynamic BFT system incorporates a finite set of n replicas $\Pi = \{p_1, p_2, \ldots, p_n\}$, referred to as the universe. At any given moment, a dynamic BFT group comprising a subset of Π manages requests dispatched by clients to maintain the global consistency of operation logs. The system is leader-based [46]. Within a period referred to as a view v, a replica in the group serves as the leader to drive the protocol for one or more views. Each replica $p_i \in \Pi$ possesses a public/private key pair (pk_i, sk_i), where pk_i serves as a unique identifier known to all replicas in the set Π.

In contrast to static BFT, where the set of replicas remains constant, a dynamic BFT model allows replicas to join and leave the system dynamically, necessitating the introduction of the notion of configurations to represent the successive membership of a dynamic BFT group. A configuration, numbered by a monotonically increasing integer c and initialized as 0, specifies the membership M_c of a specific moment (i.e. the group of replicas in that configuration). A replica p is considered to be in configuration c if $p \in M_c$. If p has installed configuration c but has not installed any other configuration $c' > c$, then we refer to the current configuration of replica p as c. We also refer to c as the latest configuration of the system. if at least one non-faulty replica has installed c and no non-faulty replica has installed a $c' > c$. If two configurations c_1, c_2 have the same membership, denoted as $c_1 \approx c_2$.

We consider a partially synchronous network [29] in the system, in which a valid proposer must await vote messages from other validators before proposing a block containing requests. After some unknown global stabilization time (GST), all messages transmitted between non-faulty replicas are guaranteed to be delivered within the time span specified by Δ. Our system model considers an adversary capable of dynamically corrupting a subset $F \subseteq \Pi$ of replicas, with the size of F limited to $|F| \leq f$, where $f = \frac{n-1}{3}$. These compromised replicas may exhibit Byzantine faults, characterized by arbitrary disruptive behaviors, including protocol violation or non-response. The adversary may intercept network communication but lacks the capacity to impede or alter messages during point-to-point transmission. Additionally, it must induce the GST event after an indefinite but finite time period. Suppose that there are f_c replicas of configuration c under the control of the adversary, for each c and corresponding membership M_c, we introduce the following definitions:

Definition 1: A configuration c is correct if $f_c \leq \frac{|M_c|-1}{3}$.

c is faulty if it is not correct. Our consensus mechanism operates under the foundational assumption that there exists a non-empty subset of replicas, $L \subseteq M_0$ satisfying $|L| > \frac{f-1}{2}$, where each replica in L is guaranteed to be correct and never leave the system. Some, but not all, replicas in L are permitted to experience a temporary crash fault, provided they can subsequently be recovered to a correct state. Leadership rotation

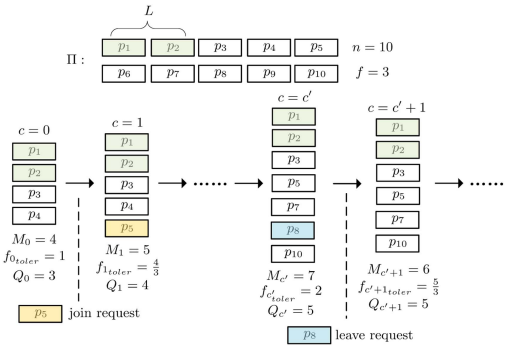


Fig. 1. System Model.

occurs exclusively within L as the system view changes. The setting of L serves to preclude that up to f malicous replicas will not form a quorum in some configuration, and to prevent correct replicas from being fooled into conflicting consensus. These two stipulations constitute the sole assumptions regarding membership correctness in our model.

Let $f_{c_{toler}} = \frac{|M_c|-1}{3}$, a quorum of replicas in c is defined as a majority set consisting of $\lceil \frac{|M_c|+f_{c_{toler}}+1}{2} \rceil$ replicas, the size of which is denoted by Q_c. For any proposal or request in c to be considered valid and subsequently committed, it must garner valid votes from at least Q_c replicas to construct a proof. A valid request is one that conforms to the protocol's format and authentication requirements. For membership requests specifically, the validity criteria are detailed in Section V. A valid vote is a vote message with a correct cryptographic signature from a replica that is responsible for the request being voted on (replica responsibilities will be defined subsequently). For a given view v and configuration c, we define as follows:

Definition 2: $\langle c, v \rangle$ is valid if, based on c in v, quorums of valid votes can be collected for valid requests.

Definition 2 is more reasonable than Definition 1 when $c > 0$ since it only considers the final collected votes. A malicious replica can disguise itself by casting a valid vote, so $\langle c, v \rangle$ valid does not necessarily mean that c is correct. A visual representation of our system model is illustrated in Fig. 1.

B. Changes in the Group Membership

In addition to regular BFT requests, our system incorporates membership requests, which include join requests for adding new replicas and leave requests for removing existing replicas from the group. These requests are treated in the same way as regular requests, implying that join and leave requests within a view are only converted to formal configurations when they are valid (the validity criteria for membership requests will be elaborated in Section V) and when all other requests within the same view meet the commit conditions. When referring to a request later, it may either indicate a membership request or a regular request. We also do not taking into account the initiator of membership requests (join or leave) or the reasons behind them, which are possibly faulty. For any request, the membership M_c of c in which the quorum votes is collected for the first time are responsible for that request (**also implies that the request is based on** c).

Our protocol follows the three-phase paradigm (two confirmation phases prior to commitment) of leader-based BFT protocols. To pipeline membership requests along with regular requests, we introduce two temporary memberships: M_{high} and M_{valid}. We denote M_{valid} as the latest valid temporary membership known to replicas. For a proposal m in view v of configuration c, if $\langle c, v \rangle$ is valid, the membership requests contained within are first applied based on M_{valid} to obtain M_{high}. If M_{high} is valid and m receives another confirmation, M_{high} is assigned to M_{valid}. If m satisfies the commit condition, M_{valid} transitions to $M_{c'}$, marking the formal shift to the new configuration c', and all membership requests are formalized. Intuitively, M_{high} represents the tentative membership obtained after processing join/leave requests from a proposal that has received its first confirmation, while M_{valid} represents the membership after the second confirmation. At system initialization, both temporary memberships are set to the initial configuration membership, i.e., $M_{valid} = M_{high} = M_0$. By mirroring the three-phase progression of regular requests with three-stage membership transitions, this approach facilitates the processing of member requests at the same pace as regular requests.

Specifically, upon processing a join request for replica p_i at replica p_j based on M, the membership for p_j becomes $M' = M \cup \{p_i\}$. Conversely, upon executing a leave request for p_i at p_j based on M, its membership becomes $M'' = M - \{p_i\}$. For two configuration memberships M_i and M_j,we use M_j/M_i to represent the set of replicas that in M_j but do not in M_i.

Replicas initiating join requests start voting upon entering temporary configurations (M_{high}, M_{valid}), and their votes construct proofs of validity for the temporary configurations. Replicas initiating leave requests do not vote on new proposals after leaving the formal configuration but must confirm or commit pending proposals they were responsible for before exiting. Non-faulty replicas follow these rules, while malicious replicas might try to leave prematurely or refuse to vote in temporary configurations. The system also includes a configuration auto-transition mechanism that attempts to diminish the quantity of malicious replicas below the Byzantine tolerance: leader marks replicas with inconsistent behavior through configurations after a timeout and automatically initiate leave requests for them, aimming to address the unique liveness issue (see Section III-E) due to weaker assumptions.

C. Essential Properties

Following Static BFTs, we first focus on the Byzantine Atomic Broadcast (BAB) problem [47], allowing replicas to agree on the sequence of messages (requests) required for SMR. The following definition is provided, some of the properties of which are adapted in conjunction with the dynamic BFT model and do not fully align with the static BFT:

Definition 3 (Byzantine Atomic Broadcast): All correct (non-faulty) replicas agree on the sequence of messages broadcasted and committed, despite presence of Byzantine faults. A dynamic BFT protocol solves Byzantine agreement if it satisfies the following properties:

- *Agreement:* If a correct replica $p \in M_c$ commits a request m based on c, then all correct replicas in configuration c eventually commit the same m.
- *Total Order:* If a correct replica commits a request m before m', then no correct replica commits m' before m.
- *Validity:* If a correct client or replica initiates a request m, then correct replicas in some configuration c will eventually commit m.
- *Integrity:* A correct replica commits at most once in a view, regardless of the requests.

Total order and integrity are **safety** properties, while the other two guarantee **liveness**. In addition to the properties of BAB, we also specify three additional properties to ensure correctness in the presence of dynamic membership changes.

- *Isoconfigurability:* If a correct replica $p_i \in M_c$ commits m based on configuration c and a correct replica $p_j \in M_{c'}$ commits m based on c', then $c = c'$.
- *Clarity:* A correct client or replica initiating a request m can eventually learn about the correct membership of the configuration c based on which m is committed. A client will also receive a response from a correct replica in M_c.
- *Configuration discovery:* A correct replica will eventually learn about a continuous and verifiable configuration history of the system up to the latest valid configuration c. The configuration history is totally ordered.

Isoconfigurability ensures replicas commit the same requests in the same configuration, preventing potential inconsistencies due to membership changes, which is is crucial for dynamic BFT systems (each request is assumed unique. If the same content initiated multiple times, it will be treated as a different request). Clarity allows the initiator of m to identify the members in c responsible for m and determine the final result based on responses from correct replicas in c and whether m have satisfied the commit condition. Dynamic BFT systems require this property because the latest membership of the system is constantly changing due to membership requests, and it is not possible to ensure that the configuration is still in c when a request m based on c is committed.

Static BFTs typically necessitate the inclusion of state transfer mechanisms to update downed or lagging replicas to the highest valid view. This is further enhanced in dynamic BFT, where replicas falling behind or initiating join requests must be kept up to date with the latest configuration via a state transfer mechanism, clarifying the current membership and voting threshold. A lagging replica's vote may be determined as invalid and thus be regarded as a fault. Configuration discovery enables replicas to ascertain contiguous and the latest system membership, thereby ensuring that correct replicas are in agreement across configuration histories. To achieve this, additional protocols must be introduced that hinge on how the latest configuration is discovered and how the validity of the configuration is verified, discussed in Section V-A.

D. Comparison With Prior Works

A comprehensive comparison of our approach with Schiper's dynamic membership service [12] and the state-of-the-art dynamic BFT solution Dyno [21] is presented below, highlighting our distinctive improvements and objectives.

Our model, like Schiper's, separates the protocol from the membership service but adopts security definitions that addresse Byzantine failures, unlike Schiper's focus on uniform broadcast primitives under crash fault tolerance. Compared to Dyno, our model assumes weaker membership correctness. Dyno assumes either correct replicas after a configuration transition still forms a quorum or there exists at least $F = \max(f_c) + 1$ correct replicas across all configurations, which is impractical. We only require that there exists more than $\frac{f-1}{2}$ correct replicas never leave the system. Our goal is to achieve the standard Agreement property under weaker assumptions regarding membership correctness. However, this approach introduces challenges that are not present in Dyno, which we will discuss in the next subsection.

We introduce temporary configurations and define replica responsibilities for a more explicit request handling, avoiding potential liveness issues seen in Dyno. Quorum proof for requests is constructed only among responsible replicas, and correct replicas initiating leave requests must complete their responsibilities before leaving the system. A non-blocking configuration discovery protocol is also designed to facilitate rapid participation of new replicas, preventing them from being stuck in configuration discovery when configuration is high or transits frequently. These techniques aim for efficient configuration transitions and stable performance, mitigating performance fluctuation due to membership request seen in Dyno and static BFT protocols supporting reconfiguration [23].

With regard to the properties, unlike Schiper's separate definition, we do not distinguish between regular requests and membership requests in terms of total order. Both types of requests are treated equivalently. Our approach aligns with Dyno's criteria for configuration delivery and discovery. However, while aim to ensure consistency in interactions between clients and the system, Dyno's consistent delivery property only requires the client eventually delivery a correct response consistent with the state in some configuration when m is committed by someone. However, the same state does not imply the same configuration, and it may even be the case $c' \approx c$. Also, there is no requirement as to the reply source, leaving open the possibility that the response to m received from a replica that is not a member of c or a replica that was in c but has left and rejoined after c. To address this, we augment the model by introducing, as an alternative, the Clarity.

E. Core Challenges

In our dynamic BFT model, any replica in Π can initiate join or leave requests. With weaker assumptions about membership correctness, we must address scenarios where the number of malicious replicas f_c in configuration c and view v exceeds the tolerable threshold $f_{c_{toler}}$ (note that f_c will not form a quorum in any configuration due to the presence of $|L|$ correct replicas). Two fundamental cases lead to $f_c > f_{c_{toler}}$:

1) $f_{c-1} \leq f_{c-1_{toler}}$ holds in configuration $c-1$, but the joining of faults causes $f_c > f_{c_{toler}}$ in c.

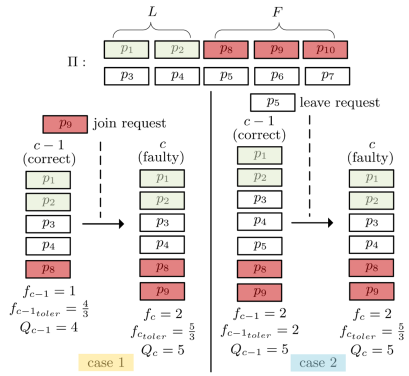


Fig. 2. Two cases that may lead to $f_c > f_{c_{toler}}$, $n = 10$.

2) $f_{c-1} \leq f_{c-1_{toler}}$ holds in configuration $c-1$,, but correct replicas initiate leave requests and transition to c. Due to the presence of faults in $c-1$, $f_c > f_{c_{toler}}$ in c.

Fig. 2 illustrates these two cases. Even if $f_c > f_{c_{toler}}$ in c, malicious replicas might not misbehave immediately, keeping $\langle c, v\rangle$ valid. Subsequently, additional faults can join, transitioning to $c+1$ and view v', where $f_{c+1} > f_{c+1_{toler}}$. At this point, malicious replicas can render $\langle c+1, v'\rangle$ invalid, constructing consecutive faulty configurations. Although the set L limits malicious replicas from causing conflicting consensus within the same view, it is evident that when $f_c > f_{c_{toler}}$, malicious replicas can invalidate $\langle c, v\rangle$, causing a unique liveness issue where correct replicas are unable to continue reaching consensus not owing to a leader failure. A carefully designed configuration auto-transition protocol is needed, not just a regular view-change mechanism.

Furthermore, the timing of a replica's entry into a formal configuration or exit from the system must not coincide with the initiation or commit of a membership request. Consider a leave request scenario: suppose the latest configuration is c, and the latest view is v. If a correct replica p_i initiates a leave request and leaves the system after c, in view $v+1$, the leader must not only propose a new proposal but also pre-commit to requests (proposal) m from v, as m was based on c and must be committed based on c. Since p_i has left, this may prevent committing m by all correct replicas in c, failing to achieve the standard Agreement property. Thus replicas initiating membership requests may have to vote outside of the formal configuration, which introduces two design challenges: first, Defining the voting period outside the formal configuration to satisfy Agreement without extra work before entering the formal configuration or before leaving the system; second, determining how they should vote and how their votes are handled, since replicas leaving the formal configuration should not deal with any new requests until they leave the system, and joining replicas in the temporary configuration only vote to prove temporary configuration validity.

IV. Data Structures and Phases

Messages: Let $\langle m\rangle_p$ denote a message m signed by replica p using sk. To generate a fully functional dynamic BFT system, we define the following types of messages.

1) A proposal message $\langle \text{PROPOSE}, node, v, c, M_c, [proof]\rangle_l$ contains a $node$ proposed, a sender's current view number v automatically stamped, a configuration c and the corresponding membership M_c, and a set of proofs $[proof]$.

2) A vote message $\langle \text{VOTE}, v, c, node_h\rangle_p$ contains a sender's current view number v automatically stamped and a configuration c. $node_h$ is the hash of the referred node.

3) A membership request (message) $\langle \text{JOIN}, pk\rangle_p$ or $\langle \text{leave}, pk\rangle_p$ contains a sender's public key pk.

4) A transition vote message $\langle \text{TRANS}, seq, v, c\rangle_p$ comprises an additional field seq indicating the number of consecutive timeouts that have been experienced. After a timeout, the replica sends a transition vote to trigger the configuration auto-transition protocol.

5) An auto-trans message $\langle \text{AUTO}, node, v, c, M_c, proof\rangle_l$ contains a $node$ containing only membership requests, a sender's current view number v automatically stamped, a configuration c and the corresponding membership M_c, and an auto-transition proof $proof$ composed of transition votes.

6) A configuration discovery message $\langle \text{CDIS}, c_{start}, c_{end}\rangle_p$ comprises a missing and attempted-to-be-acquired configuration interval from c_{start} to c_{end}.

7) A discovery message $\langle \text{DIS}, c_{start}, subconf\rangle_p$ responding to a configuration discovery comprises a historical configuration segment $subconf$, and a c_{start} indicating the commencement of the aforementioned segment.

Node: A node $node$ contains a (or a batch of) requests $node.reqs$. Both membership requests and regular requests are packaged together in $reqs$ when broadcasting proposals.

Proof: A proof $proof$ for a $node$ is a data type that includes a type $proof.type \in \{\text{HIGH}(ship), \text{VAL}(ship), \text{CON1}(c), \text{CON2}(c), \text{COM}(c), \text{AUTO}(ship)\}$, a hash $proof.node_h$, and a set of votes $proof.votes$ combining the votes of $node$ in view $proof.view$ by a quorum of replicas from the membership determined by $proof.type$: for $\text{CON1}(c)$, $\text{CON2}(c)$, and $\text{COM}(c)$, the votes are from replicas in M_c; for $\text{HIGH}(ship)$, $\text{VAL}(ship)$, and $\text{AUTO}(ship)$, the votes are from replicas in $ship$ (these types are associated with a specific membership list rather than a configuration index, as the members have not yet been formally assigned a configuration number).

Historical Configurations: All replicas maintain a continuous list of configurations $conf[]$ locally. For each $c > 0$, $conf[c]$ records the membership $conf[c].ship$ of c (i.e., M_c), the node $conf[c].node$ containing the membership requests that changed from $c-1$, and the proof $conf[c].proof$ from quorum replicas responsible for $conf[c].node$ in the commit phase or Q_c replicas in c from the configuration auto-transition protocol. Correct replicas must ensure the list is continuous and verifiable, and perform configuration discovery upon receiving any non-continuous valid configurations.

Member Marking Table: A member marking entry contains a triple $\langle p, res, time\rangle$, where p is the marked replica (also used as the key for lookup), res is the remaining marking duration, and $time$ is the number of times a replica has been marked. This table is used to mark replicas that exhibit inconsistent behavior through configurations. Any membership request initiated by replicas in the table with res (decremented after a successful

Initialization: Π, M_0, L
Local variables: ▷ *It will be prefixed when referring to a local variable.*
$view \leftarrow 1$ ▷ *View number.*
$c \leftarrow 0$ ▷ *Current configuration.*
$M_{high} \leftarrow M_0$ ▷ *Upcoming membership.*
$M_{valid} \leftarrow M_0$ ▷ *The latest valid temporary configuration.*
$conf \leftarrow Vec::new()$ ▷ *Historical configurations.*
$node_{prep} \leftarrow \perp$ ▷ *node in the prepare phase.*
$proof_{pre} \leftarrow \perp$ ▷ *Proof for the node in the pre-commit phase.*
$proof_{com} \leftarrow \perp$ ▷ *Proof for the node in the commit phase.*
$pool_m \leftarrow Vec::new()$ ▷ *Membership requests pool.*
▷ *Member Marking Table, maintained locally by replicas from L.*
$mmtable \leftarrow HashMap :: new()$ ▷ $HashMap\langle p_i, (res, time)\rangle$

Fig. 3. Local variables.

leadership proposal) greater than 0 will be ignored. The table is maintained locally by each replica from L and does not participate in communications.

Phases: BFT protocols generally mandate that a request pass through several confirmation phases prior to commitment. Notable instances include the PBFT model [22], [48] (prepare and commit phases); the HotStuff model [25], [32] (prepare, pre-commit, and commit phases); and the Algorand model [6] (various "step" phases before reaching the "final" phase). Our model adopts a pattern similar to HotStuff and streamline the distinction between various phases. A valid proposal (node) proposed in the normal state protocol based on c within a view must be confirmed by Q_c replicas in M_c in the subsequent two views before it can be committed.

V. HYDRA

We now present Hydra, focusing on how membership and regular requests are processed. It functions in a series of views, each with a monotonically increasing number and a unique dedicated leader from L known to all. Clients send command requests to replicas, and waits for responses from at least one correct replica to complete a request. Such a response contains (i) the node that includes the client's request and (ii) the corresponding COM(c) quorum proof. Given this response, the client runs the same configuration discovery procedure as replicas (detailed in Section V-A) to obtain the historical configurations up to c (thus learning M_c), verify the validity of c, and then check the validity of the quorum proof. For the most part, we omit the client from the discussion.

The basic utilities and local variables are specified in Fig. 3–4. Besides intrinsic functions, two check functions are required: Vrf to ascertain whether a proof is constructed in accordance with a specific type or configuration for a given node. isValHis to ascertain the validity and continuity of a segment of historical configuration. Note that the verification of the signature has been omitted from the pseudocode.

A. Configuration Discovery

Configuration discovery protocol enables replicas to ascertain contiguous and the latest system membership. A lagging replica or joining replica must be simultaneously kept up to date

implementation of *Proof*
function new ($type, node, view, votes$) ▷ $votes : HashMap\langle pk, vote\rangle$.
return $Self\{type, node, view, votes\}$ ▷ *self refers to struct itself.*
function Vrf ($self, view, node_h, type$) ▷ *Proof verification*
if $self.node_h = node_h \wedge self.view = view$ **then**
$vts \leftarrow \{vt.pk | vt \in self.votes\}$
match ($self.type, type$): (HIGH($ship$), HIGH($ship'$)) | (VAL($ship$), VAL($ship'$)) | (AUTO($ship$), AUTO($ship'$)) => {**return** $ship = ship' \wedge vts \subseteq ship \wedge |vts| = Q_{ship}$}
(CON1(c), CON1(c')) | (CON2(c), CON2(c')) | (COM(c), COM(c')) => {**return** $c = c' \wedge vts \subseteq M_c \wedge |vts| = Q_c$}
return *false*
implementation of *Node*
function new ($reqs$) **return** $Self\{reqs\}$
function isValHis ($subconf$, c_{start})
$i \leftarrow 0$, $M_{c_i} \leftarrow M_{c_{start}}$
while $i < |subconf|$ **do** $ship \leftarrow subconf[i].ship$
if $L \not\subset ship$ **then return** false ▷ *Prevention of forgery*
if $M_{c_i} \cup \{p|$ for $\langle$JOIN, $pk\rangle_p$ in $subconf[i].node.reqs\}$ - $\{p|$ for $\langle$LEAVE, $pk\rangle_p$ in $subconf[i].node.reqs\}$=$ship$ **then**
$proof \leftarrow subconf[i].proof$
if $proof$.Vrf($subconf[i].node, proof.view,$ COM(c_i))$\vee$ $proof$.Vrf($subconf[i].node, proof.view,$ AUTO($ship$)) **then**
$M_{c_i} \leftarrow ship$, $i \leftarrow i+1$, **continue**
return false
return true

Fig. 4. Basic utilities.

1: **function** ConfDis (c_{start}, c_{end}) ▷ *Non-blocking function*
2: wait until the existing Δ_{dis} timeout.
3: broadcast $\langle$CDIS, $c_{start}, c_{end}\rangle_p$
4: start a new confdis timer Δ_{dis}
5: **upon** receiving $\langle$DIS, $c_{start}, subconf\rangle_{p'}$
6: **if** $|subconf| = c_{end} - c_{start}$ **then**
7: **if** isValHis($subconf, c_{old}$) **then**
8: $self.conf[c_{start} : c_{end}] \leftarrow subconf$
9: $cancel\_timer(\Delta_{dis})$ **return**
10: **upon** timeout(Δ_{dis}) do ConfDis($0, c_{end}$)
11: **as** a replica p **do**
12: **upon** receiving configuration $c', M_{c'}$ from any message: $c' > self.c$
13: wait for vote from different replicas in either vote messages or votes in proposal: $V \leftarrow \{vote | vote.c = c'\}$ until $|V| = f + 1 \vee \exists v \in V$ from $p' \in L$
14: **while** $|self.conf| - 1 < c'$ **do** $self.conf.push([])$
15: $c \leftarrow self.c, self.c \leftarrow c', self.conf[c'] \leftarrow M_{c'}$ ▷ *Catch up first.*
16: ConfDis(c, c') ▷ *Runs in the background*
17: **upon** receiving $\langle$CDIS, $c_{start}, c_{end}\rangle_{p'}$
18: **if** Δ_{dis} exists **then return**
19: **if** $c_{start} < c_{end} \leq |self.conf| - 1$ **then**
20: reply with $\langle$DIS, $c_{start}, self.conf[c_{start} : c_{end}]\rangle_p$

Fig. 5. Configuration discovery.

with the latest configuration. Correct replicas must ensure their configuration list is valid, continuous and verifiable.

The pseudocode is shown in Fig. 5. The core function ConfDis allows replicas to obtain any segment of the historical configurations from others and replace their non-contiguous parts with the received valid configurations (lines 5–9). When a replica sends a configuration discovery message, it needs to start a confdis timer (line 4) and cannot respond to any configuration discovery-related communication before the timer expires (line 18). This prevents replicas undergoing configuration discovery from sending their non-contiguous configurations to other requesting replicas, reducing unnecessary overhead, and prevents

malicious replicas from attempting to disrupt the operation of correct replicas by sending many CDIS messages within a short period.

If the replica cannot obtain a valid configuration segment before the timer expires, it indicates that the existing configuration list may contain erroneous segments, requiring configuration retrieval from the beginning (line 10, in fact, it's just a precautionary setup. Since $|L|$ correct replicas exist and they will not all crash, any valid requested segment can be obtained from at least one correct replica. Usually occurs due to a short timeout setting).

Our configuration discovery algorithm is non-blocking, and can run in parallel with the consensus. Whenever a non-continuous but valid configuration received (line 12–13), the replica first skips the non-contiguous part of the historical configurations and continues the consensus with the highest known valid configuration (lines 14–15). Subsequently, it gradually fills in the non-continuous parts (lines 16). As the protocol progresses to higher configurations, it takes longer to complete the configuration list through configuration discovery. If replicas were to block and wait for the completion to perform the next step (i.e. vote or propose), it may impact their voting in configurations where they have membership, and potentially encounter non-continuous configurations again if another configuration transition occurs when blocking (especially at high transition frequencies), severely affecting consensus efficiency. Furthermore, the non-blocking feature allows (new) replicas initiating join requests to rapidly participate in the consensus process, ensuring transition efficiency.

In principle, a PBFT-like checkpointing garbage-collection mechanism could be adopted to prune historical configurations. However, unlike PBFT's logs, our discovery mechanism depends on a continuous and verifiable configuration chain to let temporarily offline or newly joining replicas validate the latest configuration. Safe reclamation would thus require that a "stable" checkpoint configuration be locally materialized by all correct replicas before discarding earlier records. Otherwise, a correct replica that was away longer or never joined before (whose highest known valid configuration is lower than the checkpoint configuration) would be unable to retrieve and verify continuity to the latest configuration with the existing configuration discovery mechanism. Since replicas cannot tell which peers are correct at runtime, the only conservative criterion is to wait for all current members, which allows Byzantine replicas to stall pruning by not participating. In addition, if some replicas crash and lose history, they can still reconstruct and verify the full chain up to the current configuration via non-blocking configuration discovery. Given that at most one formal configuration can be installed per view which stores only a membership list, a quorum proof, and a pointer to a specific node, we do not aggressively garbage-collect configuration history here. Nevertheless, we consider it to be a potential area for future research.

B. M_{high}, M_{valid}, and Replica Responsibility

In Hydra, if a proposal m in configuration c and view v contains membership requests, and $\langle c, v\rangle$ is valid, we first execute the membership requests based on M_{valid} to obtain M_{high}. For the next proposal, if M_{high} is valid and m receives another confirmation, it becomes the new M_{valid}. If M_{valid} is valid and m satisfies the commit condition, the configuration transitions to a new formal c' such that $M_{c'} = M_{valid}$. This unifies the phases for requests being commit/transited to a formal configuration, allowing the membership requests to be pipelined with regular requests. To implement this, we establish temporary configurations for M_{valid} and M_{high}. In c, the leader not only needs to collect votes from replicas in M_c but also from replicas in M_{valid}/M_c and M_{high}/M_c to construct quorum (Q_{valid}, Q_{high}) proofs to prove the validity of these two temporary configurations to replicas in c.

Note that the votes of joining replicas who only in the temporary configurations are only used to construct proofs of validity for the temporary configurations. Such replicas need not participate in the confirmation and commitment of proposals in views associated with the temporary configuration. Replicas initiating leave requests still need to be responsible for the newly initiated proposals until they leave the formal configuration. After that, they simply stay in the system, confirm or commit any pending proposals they were responsible for before exiting, and then they can leave.

We require correct replicas to follow these rules, as they are closely related to maintain the Agreement property. Specifically, if a proposal proposed in c satisfies the commit condition, it must be committed by all correct replicas in c; otherwise, no correct replicas in c should commit it.

To better illustrate the aforementioned process, Fig. 6 provides a visual example. The join request of replica p_5 is packaged into the proposal m_2 in view 2. p_5 then enters M_{high} in view 3, and M_{valid} in view 4. Finally, in view 5, p_5 enters the formal configuration. View 5 is also the commit view for proposal m_2, ensuring consistency among all requests within the same view. p_5 needs to vote on proposals m_3 and m_4, but it is not responsible for confirming or committing them. Its responsibility starts from proposal m_5 onwards.

The leave request of replica p_4 is packaged into the proposal m_3 in view 3. p_4 remains responsible for proposals in views 4-5 within the temporary configuration. After leaving the formal configuration, p_4 continues to confirm or commit any pending proposals it was responsible for in view 6-8 before exiting, and then leaves the system.

Configuration auto-transition: To handle the unique liveness challenge discussed in Section III-E. Hydra employs a configuration auto-transition protocol that combines configuration auto-transition and view changing (a precautionary setup, triggered when multiple consecutive auto-transitions fail). The protocol is triggered on timeout, usually indicating the leader cannot create the necessary proofs to broadcast a new proposal. Since leaders are elected in L, this typically means the number of faults has exceeded the tolerable threshold $f_{c_{toler}}$. An auto-transition is performed first to mark replicas with inconsistent behaviour and automatically initiate leave requests for them to leave the current configuration. The marking rules follow the principle of '**better to kill by mistake than to let go**', aiming to recover from an invalid $\langle c, v\rangle$ to a valid $\langle c', v'\rangle$ to facilitate consensus.

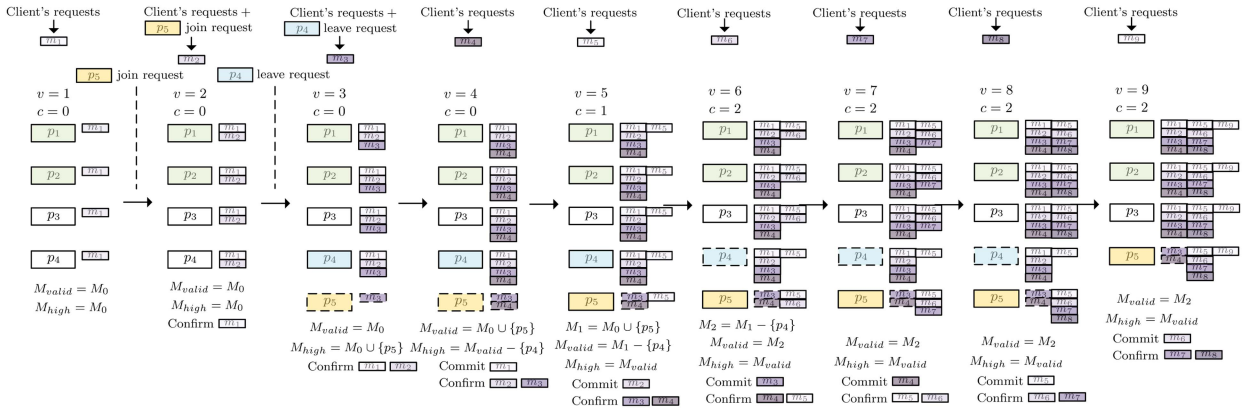


Fig. 6. An example of configuration transitions and replicas responsibilities (all configurations are valid in this example).

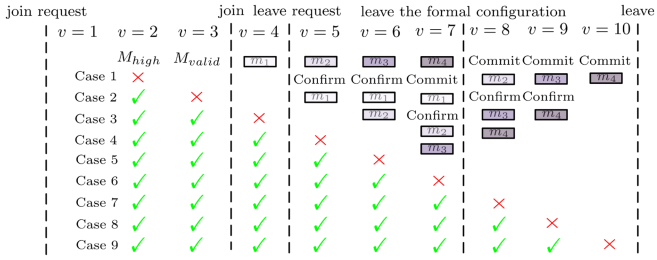


Fig. 7. Scenarios of misbehavior.

Fig. 7 depicts scenarios where malicious replicas misbehave (with ✓ for correct behavior and × for misbehavior). These scenarios are categorized based on the replica's configuration status: (1) Temporary Configurations: Prior to joining the formal configuration, replicas misbehave (Cases 1-2). If a HIGH or VAL proof cannot be constructed before the timeout, the leader marks the non-voting replicas in M_{high}/M_c or M_{valid}/M_c. (2) Formal configuration (Cases 3-6): During each phase, if the proof cannot be constructed before the timeout, the leader marks all non-voting replicas in the corresponding configuration. (3) After leaving the formal configuration but before leaving the system, replicas misbehave (Cases 7-8). Similar actions as in (2) are taken if required proofs for pre-commit or commit phases cannot be constructed. Case 9 is exceptional, where a malicious replica's failure to commit a proposal or execute a request is undetectable by the leader but does not compromise system safety or liveness.

After marking and removal, the auto-transition protocol must ensure continuity and verifiability from the original configuration, manage replica misbehavior, and handle uncommitted (pending) requests. If multiple consecutive auto-transitions fail, it may be inferred that the leader may have crashed, necessitating a view change.

C. Protocol Specification

Fig. 8 presents the pseudocode for normal state protocol, described as an iterated view-by-view loop. In each view, replicas execute phases successively according to their roles, described as a succession of "as" blocks. A replica can have multiple roles (e.g., the leader is also a replica), and "as" blocks for different roles can execute concurrently.

We start with the leader. A view's leader first collects votes from replicas in the previous view and constructs a proof list containing up to five proofs for proposals in different phases and temporary configurations (lines 6–12. If no proposal is in the pre-commit or commit phase, the corresponding proof can be vacant). A replica's vote can be used for constructing multiple proofs (e.g., a replica's vote for a new proposal also serves as confirmation for proposals in the pre-commit and commit phase if it is responsible for). The leader then packages requests from clients and in its own membership requests pool into a newly created node, creates a proposal, and broadcasts it along with the proof list (line 13). The new proposal then enters the prepare phase.

As a responsible replica (line 16), upon receiving a valid proposal from the current leader, it sends its vote to the next leader (line 18). It is essential to record the state before processing the proof list (lines 21–23) and uses it as the reference for subsequent state updates to avoid data chaos. If a valid proof for a node in the prepare phase is contained in the list (line 24), the membership requests in the node are applied on M_{valid} to obtain a new $self.M_{high}$, and $self.proof_{pre}$ is also updated to the new one. If both a proof for the node in the pre-commit phase matching $proof_{pre}$ and a proof for the corresponding temporary configuration (M_{high}) are contained (line 27), the node enters the commit phase and the latest valid temporary membership is updated (line 28). If both a proof for the node in the commit phase matching $proof_{com}$ and a proof for the corresponding temporary configuration (M_{valid}) are contained (line 29), replica executes requests in the node (if the replica is responsible for the proposal), transits its M_{valid} to a new formal configuration (lines 30–33) and update its historical configurations.

The pseudocode for membership request issuing and processing is shown in Fig. 9. As a new replica that wants to join, it randomly sends a join request to a replica in L (line 2). It then updates its latest known state using the received valid proofs and enter the normal state protocol (lines 3–5). As a replica that wants to leave, it also randomly sends a leave request to a replica in L (line 7) and continues with the normal state protocol until it is no longer responsible for any proposal and then leaves the system.

1: **for** $self.view \leftarrow$1,2,3,... **do**
2: $c' \leftarrow$ **match** $self.proof_{pre}.type$ {CON1(c) => c,_ => $null$}
3: $c'' \leftarrow$ **match** $self.proof_{com}.type$ {CON2(c) => c,_ => $null$}
4: **as** the LEADER($self.view$) **do** ▷ $M_c \leftarrow conf[c].ship$
5: $proofs \leftarrow Vec$::$new()$
6: wait for all messages: $Msg \leftarrow$
$\{\langle$VOTE$, v, c, node_h\rangle_p | v = self.view \wedge c = self.c\}$ until
there are Q_c votes: $V_{\text{CON1}} \leftarrow \{vt \mid vt.p \in M_{self.c} \wedge vt \in Msg\}$
there are $Q_{c'}$ votes: $V_{\text{CON2}} \leftarrow \{vt \mid vt.p \in M_{c'} \wedge vt \in Msg\}$
there are $Q_{c''}$ votes: $V_{\text{COM}} \leftarrow \{vt \mid vt.p \in M_{c''} \wedge vt \in Msg\}$
there are Q_{high} votes: $V_{\text{HIGH}} \leftarrow \{vt \mid vt.p \in self.M_{high} \wedge vt \in Msg\}$
there are Q_{val} votes: $V_{\text{VAL}} \leftarrow \{vt \mid vt.p \in self.M_{val} \wedge vt \in Msg\}$
7: $node_h \leftarrow m.node_h : m \in V_{\text{CON1}}, v \leftarrow m.v : m \in V_{\text{CON1}}$
8: $node_{h'} \leftarrow self.proof_{pre}.node_h, node_{h''} \leftarrow self.proof_{com}.node_h$
9: $v' \leftarrow self.proof_{pre}.view, v'' \leftarrow self.proof_{com}.view$
10: $proofs.push(Proof$::$new(\text{CON1}(self.c), node_h, v, V_{\text{CON1}}))$
$proofs.push(Proof$::$new(\text{CON2}(c'), node'_h, v', V_{\text{CON2}}))$
$proofs.push(Proof$::$new(\text{COM}(c''), node''_h, v'', V_{\text{COM}}))$
$proofs.push(Proof$::$new(\text{HIGH}(self.M_{high}), node'_h, v', V_{\text{HIGH}}))$
$proofs.push(Proof$::$new(\text{VAL}(self.M_{val}), node''_h, v'', V_{\text{VAL}}))$
11: $node \leftarrow Node$::$new(\{$client's commands$\} \cup self.pool_m)$
12: $self.prool_m \leftarrow Vec$::$new()$
13: broadcast $\langle$PROPOSE$, node, self.view, self.c, M_{self.c}, proofs\rangle_l$
14: **for** $(key, value)$ in $self.mmtable$ **do**
15: $value.res \leftarrow value.res - 1$ ▷ *Reduce remaining marking time*
16: **as** a replica in $M_{self.c} \cup M_{c'} \cup M_{c''} \cup self.M_{high} \cup self.M_{valid}$ **do**
17: wait for message $\langle$PROPOSE$, node, v, c, M_c, proofs\rangle_l$ from
LEADER($self.view$): $v = self.view \wedge c = self.c$
18: send $\langle$VOTE$, v, c, node_h\rangle_p$ to LEADER($self.view + 1$)
19: $self.node_{prep} \leftarrow node_h$
20: new prf_{CON1},prf_{CON2},prf_{COM},prf_{HIGH},prf_{VAL}.
21: **for** $proof$ in $proofs$ **do**
match $proof.type$ { CON1(c) =>{$prf_{\text{CON1}} \leftarrow proof$}
CON2(c) =>{$prf_{\text{CON2}} \leftarrow proof$}
COM(c) =>{$prf_{\text{COM}} \leftarrow proof$}
HIGH($ship$) =>{$prf_{\text{HIGH}} \leftarrow proof$}
VAL($ship$) =>{$prf_{\text{VAL}} \leftarrow proof$} }
22: $proof_{com} \leftarrow self.proof_{com}, proof_{pre} \leftarrow self.proof_{pre}$
23: $M_{high} \leftarrow self.M_{high}, M_{valid} \leftarrow self.M_{valid}, c \leftarrow self.c$
▷ *Start pre-commit phase*
24: **if** prf_{CON1}.Vrf($prf_{\text{CON1}}.view, prf_{\text{CON1}}.node_h$, CON1($c$)) **then**
25: $node^* \leftarrow prf_{\text{CON1}}.node$ ▷ $node_h \leftarrow hash(node)$
26: $self.M_{high} \leftarrow M_{valid} \cup \{p \mid \langle$JOIN$, pk\rangle_p \in node^*.reqs\}$ -
$\{p \mid \langle$LEAVE$, pk\rangle_p \in node^*.reqs\}$, $self.proof_{pre} \leftarrow prf_{\text{CON1}}$
▷ *Start commit phase on* $proof_{pre}.node$
27: **if** prf_{CON2}.Vrf($proof_{pre}.view, proof_{pre}.node_h$, CON2($c'$)) $\wedge$
prf_{HIGH}.Vrf($proof_{pre}.view, proof_{pre}.node_h$, HIGH($M_{high}$)) **then**
28: $self.M_{valid} \leftarrow M_{high}, self.proof_{com} \leftarrow prf_{\text{CON2}}$
▷ *Start decide on* $proof_{com}.node(commitment)$
29: **if** prf_{COM}.Vrf($proof_{com}.view, proof_{com}.node_h$, COM($c''$)) $\wedge$
prf_{VAL}.Vrf($proof_{com}.view, proof_{com}.node_h$, VAL($M_{valid}$)) **then**
30: **if** self in $M_{c''}$ **then**
31: execute $proof_{com}.node.reqs$, respond to clients
32: $node'' \leftarrow prf_{\text{COM}}.node$
33: $self.c \leftarrow c + 1, self.conf.push(\langle M_{valid}, node'', prf_{\text{COM}}\rangle)$
34: reset timer, $self.view \leftarrow self.view + 1$.

Fig. 8. Normal state protocol.

1: **as** a replica to join **do** ▷ *new replica*
2: randomly select a replica $l \in L$ to send $\langle$JOIN$, pk\rangle_p$
3: wait for $\langle$PROPOSE$, node, v, c, M_c, proofs\rangle_l$:
$\langle$JOIN$, pk\rangle_p \in node.reqs$
4: **for** $proof$ in $proofs$ **do**
match $proof.type$ { CON1(c) => {$self.proof_{pre} \leftarrow proof$}
CON2(c) => {$self.proof_{com} \leftarrow proof$}
HIGH($ship$) => {$self.M_{high} \leftarrow ship$}
VAL($ship$) => {$self.M_{valid} \leftarrow ship$} }
5: $self.view \leftarrow v$ ▷ *Enter the normal state protocol.*
6: **as** a replica to leave **do** ▷ *existing replica*
7: randomly select a replica $l \in L$ to send $\langle$LEAVE$, pk\rangle_p$
8: wait for $self \notin M_{self.c} \cup M_{c'} \cup M_{c''} \cup self.M_{high} \cup self.M_{valid}$
leave the system ▷ *Obtain* c', c'' *in normal state protocol per view.*
9: **as** a replica $\in L$ **do**
10: **upon** receiving $\langle$JOIN$, pk\rangle_p$
11: **if** $p_i \notin (M_c \cup self.M_{high} \cup self.M_{valid})$
$\wedge self.mmtable.get(p).res \leq 0$ **then**
12: $self.pool_m.push(\langle$JOIN$, pk\rangle_p)$
13: **upon** receiving $\langle$LEAVE$, pk\rangle_p$
14: **if** $p_i \in (M_c \cap self.M_{high} \cap self.M_{valid})$ **then**
15: $self.pool_m.push(\langle$LEAVE$, pk\rangle_p)$

Fig. 9. Membership request issuing and processing.

As a replica in L, the valid membership requests it receives are added to its membership requests pool (lines 10–15), which will be packaged into a node when the replica becoming the leader.

When a timeout occurs, the system triggers the configuration auto-transition protocol, as shown in Fig. 10. The leader first moves regular requests from uncommitted proposals back to the client requests pool for re-consensus after the auto-transition (line 5). It then marks replicas exhibiting inconsistent behavior according to the rules in Section V-B and automatically issues their leave requests (lines 9–21). The rule for setting the marking duration from the number of times a replica has been marked in $mmtable$ can be defined manually (here we set the former equal to the latter plus 8). Any join request from marked replicas during the marking duration will be considered invalid and ignored (Fig. 9, line 11), and the marking time decreases after a successful leadership proposal in the normal state protocol (Fig. 8, line 15).

Subsequently, the leader packages the auto-issued requests and the membership requests from uncommitted proposals into a newly created node and forms the target configuration (obtained by applying the requests in the node to the current formal configuration, as all uncommitted requests will be re-executed, and there are no temporary configurations when auto-transition is triggered; all new configurations transition from the current formal configuration) (lines 25–26). Note that this node does not include any regular requests and can be committed in a single round without causing conflicts. It is distinguished from the one in the normal state protocol in that its quorum vote needs to be collected from the replicas in the target configuration rather than the current configuration. Votes for it are only used to prove the validity of the target configuration. The leader then waits for votes from members in M_{auto} to construct a proof and propose the node.

If a quorum of votes from the new configuration can not be collected before the next timeout (line 22), the leader resets the timer, further marks replicas exhibiting inconsistent behavior in the previous auto-transition (line 24), updates the target configuration, recreates the node, and wait for votes until a proof from the new configuration is successfully constructed to broadcast the proposal. Afterward, the system return back to the normal state protocol.

After timeout, a replica performs up to $maxSeq$ consecutive auto-transitions (line 32), sending the transition vote to the current leader ($maxSeq$ can be set to 2, as after GST, auto-transitions can be completed in two attempts theoretically if the leader doesn't crash. See the proof in the appendix for details). The replica then wait until receiving a valid auto-trans message (line 36). It then verifies the configuration, updates its historical

Local variables: $remove \leftarrow Vec::new()$, $autoReq \leftarrow Vec::new()$, M_{auto}
$seq \leftarrow 0$ ▷ *Number of consecutive timeouts*
$maxSeq$ ▷ *A pre-set maximum number of consecutive auto-transitions*

1: **upon** timer timeout **as** the LEADER($self.view$) **do**
2: **if** $self.seq = 1$ **then** ▷ *V_{xxx} is the set of votes collected previously*
3: $proof_{com} \leftarrow self.proof_{com}$, $proof_{pre} \leftarrow self.proof_{pre}$
4: $M_{high} \leftarrow self.M_{high}$, $M_{valid} \leftarrow self.M_{valid}$
5: for requests in $proof_{pre}.node.reqs \cup proof_{com}.node.reqs \cup node_{prep}.reqs$, re-add non-membership requests to clients' requests pool, package membership requests into *Req*.
6: $c' \leftarrow$ **match** $self.proof_{pre}.type$ {CON1(c) => c,_ => $null$}
7: $c'' \leftarrow$ **match** $self.proof_{com}.type$ {CON2(c) => c,_ => $null$}
8: **if** $|V_{\text{CON1}}| < Q_c$ **then** ▷ *pk is the public key of p.*
9: $remove \leftarrow remove \cup \{p |\ p \in M_{self.c} \wedge !V_{\text{CON1}}.contains(pk)\}$
10: **if** $|V_{\text{CON2}}| < Q_{c'}$ **then**
11: $remove \leftarrow remove \cup \{p |\ p \in M_{c'} \wedge !V_{\text{CON2}}.contains(pk)\}$,
12: **if** $|V_{\text{COM}}| < Q_{c''}$ **then**
13: $remove \leftarrow remove \cup \{p |\ p \in M_{c''} \wedge !V_{\text{COM}}.contains(pk)\}$,
14: **if** $|V_{\text{HIGH}}| < Q_{high}$ **then**
15: $remove \leftarrow remove \cup \{p |\ p \in M_{high} \wedge !V_{\text{HIGH}}.contains(pk)\}$,
16: **if** $|V_{\text{VAL}}| < Q_{valid}$ **then**
17: $remove \leftarrow remove \cup \{p |\ p \in M_{valid} \wedge !V_{\text{VAL}}.contains(pk)\}$,
18: **for** $p \in (remove - L)$ **do match** $self.mmtable.entry(p)$: {
19: $Vacant$ => {$self.mmtable.insert((8,1))$}
20: $Occupied(res, time)$ => {$res \leftarrow 8 + (time \leftarrow time + 1)$}}
21: $autoReq.push(\langle \text{LEAVE}, pk \rangle_l)$
22: **if** $self.seq > 1$ **then**
23: **for** $p \in (M_{auto} - L) \wedge !V_{\text{AUTO}}.contains(pk)$ **do**
24: $autoReq.push(\langle \text{LEAVE}, pk \rangle_l)$
25: $node \leftarrow Node::new(\{Req \cup autoReq\})$
26: $M_{auto} \leftarrow M_c \cup \{p |\ \langle \text{JOIN}, pk \rangle_p \in node.reqs\}$ - $\{p |\ \langle \text{LEAVE}, pk \rangle_{p \text{ or } l} \in node.reqs\}$
27: wait for all messages: $Msg \leftarrow \{\langle \text{TRANS}, seq, v, c \rangle_p | seq = self.seq \wedge v = self.view \wedge c = self.c\}$ until there are Q_{auto} votes: $V_{\text{AUTO}} \leftarrow \{vt \mid vt.p \in M_{auto} \wedge vt \in Msg\}$
28: $proof \leftarrow Proof::new(\text{AUTO}(M_{auto}), node_h, self.v, V_{\text{AUTO}})$
29: broadcast $\langle \text{AUTO}, node, self.v, self.c, M_{auto}, proof \rangle_l$
30: **upon** timer timeout **as** a replica **do**
31: $self.seq \leftarrow self.seq + 1$, $v \leftarrow self.view$, $c \leftarrow self.c$
32: **if** $seq \leq maxSeq$ **then** ▷ *Do the auto-transition first*
33: send $\langle \text{TRANS}, self.seq, v, c \rangle_p$ to LEADER(v)
34: **else** send $\langle \text{VOTE}, v, c, self.node_{prep} \rangle_p$ to LEADER($v+1$)
35: reset timer, $self.view \leftarrow self.view + 1$, $self.seq \leftarrow 0$
36: wait for $\langle \text{AUTO}, node, v, c, M_{auto}, proof \rangle_l$ from LEADER(v):
37: **if** $proof.\mathsf{Vrf}(v, node_h, \text{AUTO}(M_{auto}))$ **then**
38: $newconf \leftarrow \langle M_{auto}, node_h, proof \rangle$
39: **if** $\mathsf{isValHis}(newconf, c)$ **then**
40: $self.c \leftarrow c + 1$, $self.conf.push(newconf)$
41: $self.M_{valid} \leftarrow self.M_{self.c}$, $self.M_{high} \leftarrow self.M_{self.c}$
$self.m_{prep} \leftarrow \perp$, $self.proof_{pre} \leftarrow \perp$, $self.proof_{com} \leftarrow \perp$
42: reset timer, $self.view \leftarrow v + 1$, $self.seq \leftarrow 0$
43: send $\langle \text{VOTE}, self.view, self.c, \perp \rangle_p$ to LEADER($v+1$)

Fig. 10. Configuration auto-transition protocol.

history and status (all uncommitted requests will be re-executed, and old proposals will not have a chance to be confirmed based on the original configuration), and re-enters the normal state protocol (lines 37–43). Once attempts exceeds $maxSeq$, it will go into view-changing (line 34), indicating the current leader may crash (or the timeout is too short).

The proofs for Hydra are shown in Appendix A in the supplementary material.

Complexity: A node in the normal state protocol undergoes three phases, each requiring one communication round. Thus, the time complexity here is $O(3)$, corresponding to three consecutive rounds. In the configuration auto-transition case, at most $maxSeq$ rounds are performed, leading to a worst-case time complexity of $O(maxSeq)$. Both membership request submission and configuration discovery require one communication round, with a time complexity of $O(1)$.

In the normal state protocol, the leader broadcasts to all replicas, and responsible replicas respond with a vote, resulting in a communication complexity of $O(n)$ per view. For configuration auto-transition, the same $O(n)$ complexity applies to each transition, leading to a worst-case communication complexity of $O(maxSeq * n)$. Membership request submission involves one round of communication between two replicas ($O(1)$), and configuration discovery involves one round of broadcast and response ($O(n)$).

Churn resilience: Hydra bounds effect of repeated join/leave requests (which may lead to frequent configuration transitions) in both rate and cost:

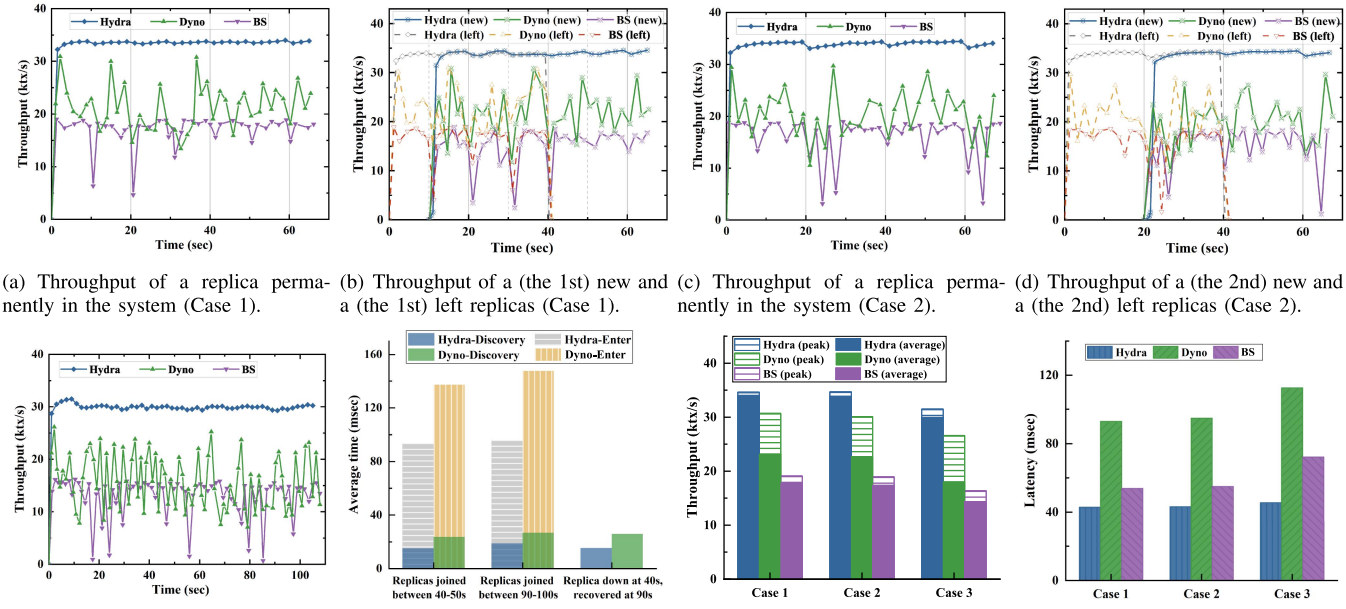
- Only (at most) one formal configuration can be installed per view because reconfiguration is finalized only upon commit. Each record in historical configurations stores only a membership list, a quorum proof and a reference pointing to a specific node, which is negligible compared to the per-view payload of the committed node (i.e., the block data in blockchain). Membership requests are small in size and are batched together with regular requests, incurring no extra phases or additional propagation overhead.
- Join requests initiated by replicas in either formal or temporary configurations are ignored. Likewise, only leave requests initiated by replicas in both formal and temporary configurations are processed. This also limits the frequency at which replicas can force configuration transitions.
- When configuration auto-transition is invoked, the leader marks replicas with inconsistent behavior and ignores their further join/leave requests for a number of views with linear backoff.
- New or recovering replicas follow the highest valid configuration and fill historical gaps in the background via non-blocking discovery, so frequent transitions do not block consensus.

As we will demonstrate later (Fig. 11(e)), Hydra maintains stability when replicas repeatedly join and leave, with minimal fluctuations in throughput.

VI. Evaluation

Implementation: We implement Hydra in about 6 K lines of Rust code, using Tokio library for asynchronous IO, ed25519 for elliptic curve based signatures and TCP connections for communication between replicas. Our client is implemented as a separate single-threaded process that send transactions to replicas at a specified rate. Replicas read configuration from both JSON files and command line arguments, and output statistics for further analysis. We also develop a module called CONFIG-GEN to automatically generate configuration files for different experiments, as well as scripts to distribute, run, and collect the results. Bash scripts are used to automate the execution of experiments, using the output from CONFIG-GEN and execute the experiments on multiple machines (servers). All of the above code is open sourced on GitHub[1] to enable reproducible results.

[1] https://github.com/BerserkRugal/Hydra



(a) Throughput of a replica permanently in the system (Case 1).
(b) Throughput of a (the 1st) new and a (the 1st) left replicas (Case 1).
(c) Throughput of a replica permanently in the system (Case 2).
(d) Throughput of a (the 2nd) new and a (the 2nd) left replicas (Case 2).
(e) Throughput of a replica permanently in the system (Case 3).
(f) Time taken for configuration discovery & from request initiation to joining.
(g) Peak throughput and average throughput in the three cases.
(h) Average latency of requests in the three cases.

Fig. 11. Performance under membership requests (BFT-SMaRt is referred to as 'BS' in the figure).

Setup: We compare Hydra with Dyno [21], a state-of-the-art dynamic BFT solution, and BFT-SMaRt [23], a high-performance BFT SMR supporting reconfiguration where membership requests are issued by a separate view manager. We use throughput as our main metric, measured as the number of transactions committed per second. When referring to latency, we mean the time a client generate a transaction to the time the transaction is committed by a replica.

We conducted experiments on a cluster of 16 Alibaba Cloud Elastic Compute Service (ECS) u1-c1m2.4xlarge instances, each with 16 virtual CPUs on a Intel Xeon Platinum scalable processor, 32 GB memory, running Ubuntu 22.04 LTS. Replicas were distributed across those instances equally. In the later part, unless otherwise specified, the batch size was 400 transactions, the transaction size was 128 bytes and the timeout was 2.5 s for each replica. The roundtrip time (RTT) between replicas was set to 10 ms, with a bandwidth of 1 Gb/s.

Performance under membership requests: Experiments were conducted in three cases.

Case 1: consensus starts with 7 replicas, three new replicas initiate join requests at 10-second intervals (starting at 10 s), and subsequently three replicas in the system initiate leave requests at 10-second intervals (starting at 40 s).

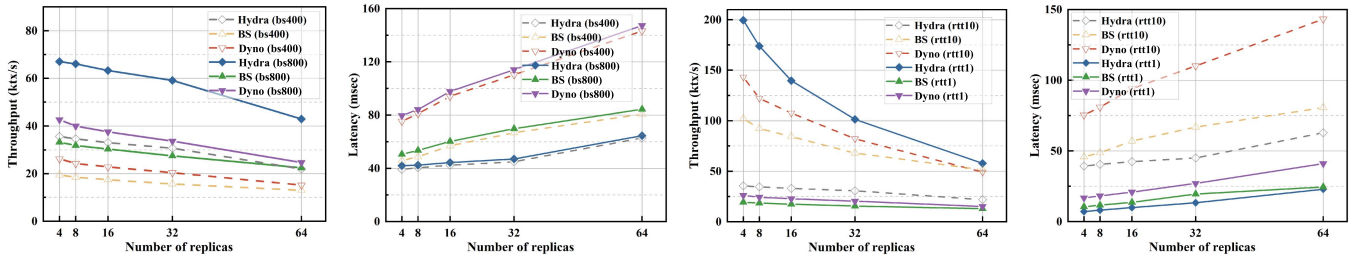
Case 2: consensus starts with 7 replicas, two new replicas initiate join requests simultaneously at 20 s, two replicas in the system initiate leave requests simultaneously at 40 s, and a new one and a existing one initiate join and leave requests simultaneously at 60 s.

Case 3: consensus starts with 20 replicas. After 20 s, replicas frequently initiate join and leave requests (new replicas repeatedly join and then leave, some existing replicas repeat the opposite action).

As shown in Fig. 11(a) and (c), replicas joining or leaving has little impact on Hydra's throughput, with a maximum of decrease from around 34.3 ktx/s to 33 ktx/s observed. Temporary configurations and replica responsibilities act as an effective buffer. Whereas Dyno sees a minimum drop from 24 ktx/s to 20 ktx/s when replicas leave, and joining has a more significant impact, with a typical decline of over 5 ktx/s. BFT-SMaRt exhibits similarities to Dyno, but occasional drops in throughput over 10 ktx/s are observed, though these recover quickly. A Minimal decrease from 18 ktx/s to 15.5 ktx/s is observed in response to membership requests. The throughput for new and left replicas (when in the system) is similar to those permanently in the system, with a slightly more fluctuation for joining replicas (Fig. 11(b) and (d)).

As shown in Fig. 11(e), frequent membership requests caused Hydra's throughput to decrease from about 31.5 ktx/s to 29.5 ktx/s, and fluctuate around 30 ktx/s. In contrast, Dyno always fluctuates broadly between 21 ktx/s and 10 ktx/s. BFT-SMaRt's range is 15 ktx/s to 10 ktx/s (ignoring occasional sharp fluctuations), usually above and below 14 ktx/s. Fig. 11(g) shows that Dyno's throughput is more volatile, with an average of 23 ktx/s in Cases 1-2, over 20% below the peak of 31 ktx/s, and over 30% in Case 3. Hydra and BFT-SMaRt are more stable, with gaps around 5% (31.5 ktx/s to 30 ktx/s) and 10% (16.5 ktx/s to 14.5 ktx/s) even in Case 3. Dyno's average latency is highest, reaching 112 ms in Case 3, while BFT-SMaRt is around 70 ms and Hydra 45 ms (Fig. 11(h)).

Fig. 11(f) shows the time taken for configuration discovery and for joining. Both protocols take similar time for configuration discovery, but Hydra has a greater advantage in downtime recovery scenario (around 15.5 ms, lower than 19 ms for joining scenario during the same period, compared to Dyno's both



(a) Throughput (various batch sizes) (b) Latency (various batch sizes) (c) Throughput (various RTTs) (d) Latency (various RTTs)

Fig. 12. Baseline performance and scalability.

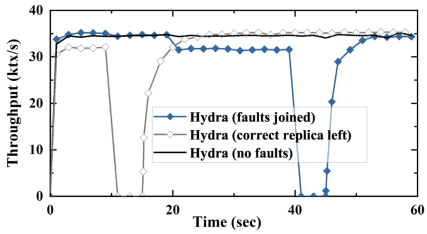


Fig. 13. Performance under faults.

scenarios are in line at 26 ms) as the discovery doesn't need to start from the initial configuration. However, entering the formal configuration takes far longer for Dyno (around 147 ms) than Hydra (around 96 ms) because Dyno must first block to complete configuration discovery before issuing a join request, while Hydra can do both in parallel.

Baseline performance and scalability: We conducted two experiments. As shown in Fig. 12(a) (throughput) and 12(b) (latency), in the first, with batch sizes of 400 ("bs400") and 800 ("bs800"), three protocols showed similar performance changes as batch size increases. Hydra achieved higher throughput and lower latency due to less communication. Dyno had higher throughput than BFT-SMaRt but lagged behind in latency. In the second experiment shown in Fig. 12(c) (throughput) and 12(d) (latency), with RTTs of 1 ms ("rtt1") and 10 ms ("rtt10"), the results obtained when comparing the three protocols are similar. However, BFT-SMaRt showed the smoothest performance degradation as replicas increased under 1 ms RTT. When $n = 64$, it surpassed Dyno in throughput and had similar latency to Hydra.

Performance under faults: We simulated the two scenarios in Section III-E that led to $f_c > f_{c_{toler}}$, as shown in Fig. 13. In the first, consensus started with 7 replicas, with 4 malicious replicas joining at 10 s intervals (starting at 10 s) in sequence ("faults joined"). In the second scenario, consensus starts with 7 non-faulty and 3 malicious replicas, with one non-faulty replica leaving at 10 s ("correct replica left"). The figure also shows Hydra's throughput with 10 non-faulty replicas ("no faults"). By utilizing the configuration auto-transition protocol, Hydra can recover from an invalid $\langle c, v\rangle$ to a valid $\langle c', v'\rangle$ to continue consensus after 1-2 timeouts, quickly restoring throughput to original levels (the target configuration c' contains fewer members after the auto-transition, because replicas with inconsistent behavior are marked and removed. The reduced number of members leads to decreased quorum sizes and network overhead, hence the slightly higher throughput. This is consistent with the scalability trend in Fig. 12). Dyno cannot handle these two scenarios (unable to continue consensus) and relies on a stronger membership correctness assumption to circumvent it.

VII. Conclusion

We present Hydra, a dynamic BFT protocol where replicas can join and leave dynamically. Hydra addresses the practical limitations of the existing dynamic BFT solution by operating under weaker assumptions regarding membership correctness while achieving standard properties. It clearly defines replica responsibilities for clarity and utilizes a non-blocking configuration discovery protocol for rapid participation. We introduce a configuration auto-transition protocol to ensure system liveness and to manage scenarios where faults join or existing correct replicas leave when faults reach the tolerance threshold—a challenge under weaker assumptions. Temporary configurations pipelines membership request alongside regular requests, ensuring smooth and efficient transitions. We have formally proven Hydra's correctness under the dynamic BFT model. Our experiments with up to 16 servers and 64 replicas demonstrate that Hydra is efficient and stable in handling membership requests, and can effectively recovers from an invalid $\langle c, v\rangle$ to a valid $\langle c', v'\rangle$.

Work limitations and future work: Looking ahead, given Hydra's assumptions continue to impose a non-trivial requirement for certain practical deployments, we envision two possible directions for exploration. First, for permissioned-but-untrusted settings, one could replace the permanent correct core with runtime safeguards that enforce a global fault bound $f' < (n-1)/3$ and a safety floor $n_{min} = 3f' + 1$ on the active configuration. If $|M_c|$ falls below n_{min}, the protocol automatically switch to a membership-only mode that process solely membership requests and resumes normal operation once $|Mc| \geq n_{min}$. The key challenge lies in designing this restricted mode to prevent adversaries from steering protocol into conflicting configurations if the same standard properties as Hydra are to be achieved. Second, when limited trusted participation is acceptable but a fixed never-leaving set is too restrictive, one could generalize

L into a rotating pool with warm standbys: L-members may leave only when a standby simultaneously joins via temporary configurations, maintaining a configured threshold for active L-members and confining leader rotation within them. We believe these directions will enhance the deployability of dynamic BFT across diverse real-world scenarios.

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Concurrency: The Works Leslie Lamport*. New York, NY, USA: ACM, 2019, pp. 203–226.

[2] E. Buchman, "TenderMint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, Eng. Syst. Comput., Univ. Guelph, Guelph, ON, Canada, 2016. [Online]. Available: http://hdl.handle.net/10214/9769

[3] S. Nakamoto and A. Bitcoin, "A peer-to-peer electronic cash system," *Bitcoin*, 4, no. 2, 2008, Art. no. 15. [Online]. Available https://bitcoin.org/bitcoin

[4] M. Belotti, N. Božić, G. Pujolle, and S. Secci, "A vademecum on blockchain technologies: When, which, and how," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3796–3838, 4th Quart. 2019.

[5] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 1432–1465, Secondquarter 2020.

[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "AlgoRAND: Scaling Byzantine agreements for cryptocurrencies," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 51–68.

[7] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 931–948.

[8] J. Kwon and E. Buchman, "Cosmos Whitepaper," *A Netw. Distrib. Ledgers*, vol. 27, pp. 1–32, 2019.

[9] N. Yang, C. Tang, Q. Zhou, and D. He, "Dynamic consensus committee-based for secure data sharing with authorized multi-receiver searchable encryption," *IEEE Trans. Inf. Forensics Secur.*, vol. 18, pp. 5186–5199, 2023.

[10] B. Hu, Z. Zhang, H. Chen, Y. Zhou, H. Jiang, and J. Liu, "Dycaps: Asynchronous dynamic-committee proactive secret sharing," *Cryptol. ePrint Arch.*, 2022. [Online]. Available: https://eprint.iacr.org/2022/1169

[11] Z. Amsden et al., "The libra blockchain," Libra Assn., Tech. Rep. 2020. Accessed: Jan. 3, 2026. [Online]. Available: https://diem-developers-components.netlify.app/papers/the-diem-blockchain/2020-05-26.pdf

[12] A. Schiper, "Dynamic group communication," *Distrib. Comput.*, vol. 18, pp. 359–374, 2006.

[13] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *Proc. 13th Pacific Rim Int. Symp. Dependable Comput.*, 2007, pp. 373–380.

[14] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, 1987.

[15] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.

[16] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in *Proc. 2nd ACM Conf. Comput. Commun. Secur.*, 1994, pp. 68–80.

[17] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y.-A. Pignolet, D.-A. Seredinschi, and A. Tonkikh, "Dynamic Byzantine reliable broadcast," in *Proc. 24th Int. Conf. Princ. Distrib. Syst.*, vol. 184, 2021, pp. 23:1–23:18.

[18] I. Coelho, V. Coelho, P. Lin, and E. Zhang, "Community Yellow paper: A technical specification for Neo blockchain," *NeoResearch*, Mar. 2019. [Online]. Available: https://neoresearch.io/assets/yellowpaper/yellow_paper.pdf

[19] Y. Wu, Z. Zhang, D. Zhu, and W. Fan, "Dynamic PBFT with active removal," in *Proc. IEEE Symp. Comput. Commun.*, 2023, pp. 1235–1241.

[20] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to Byzantine agreement," *Inf. Comput.*, vol. 118, no. 1, pp. 158–179, 1995.

[21] S. Duan and H. Zhang, "Foundations of dynamic BFT," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1317–1334.

[22] M. Castro et al., "Practical Byzantine fault tolerance," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, vol. 99, pp. 1173–1186, 1999.

[23] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with BFT-smart," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 355–362.

[24] N. Yang, C. Tang, Z. Deng, and D. He, "A gaussian reputation-based hybrid BFT consensus with a formal security framework," *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 5, pp. 5397–5414, Sep./Oct. 2025.

[25] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 347–356.

[26] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-HotStuff: A fast and robust BFT protocol for blockchains," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 2478–2493, Jul./Aug. 2024.

[27] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[28] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proc. 2nd Annu. ACM Symp. Princ. Distrib. Comput.*, 1983, pp. 27–30.

[29] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[30] E. Androulaki et al., "HyperLedger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[31] V. Buterin and V. Griffith, "Casper the friendly finality gadget," 2017, *arXiv:1710.09437*.

[32] X. Liu et al., "Dolphin: Efficient non-blocking consensus via concurrent block generation," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 11824–11838, Dec. 2024.

[33] S. Duan et al., "Dashing and star: Byzantine fault tolerance with weak certificates," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 250–264.

[34] A. Schiper and A. Sandoz, "Uniform reliable multicast in a virtually synchronous environment," in *Proc. 13th Int. Conf. Distrib. Comput. Syst.*, 1993, pp. 561–568.

[35] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *Proc. 14th Int. Conf. Distrib. Comput. Syst.*, 1994, pp. 56–65.

[36] Y. Amir and J. Stanton, "The spread wide area group communication system," Center Netw. Distrib. Syst., Johns Hopkins Univ. Baltimore, MD, USA, Tech. Rep. TR CNDS-98-4, 1998. [Online]. Available:https://ics.uci.edu/cs230/reading/spread.pdf

[37] Y. Amir, C. Nita-Rotaru, S. Stanton, and G. Tsudik, "Secure spread: An integrated architecture for secure group communication," *IEEE Trans. Dependable Secure Comput.*, vol. 2, no. 3, pp. 248–261, Jul.–Sep. 2005.

[38] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The securering protocols for securing group communication," in *Proc. 31st Hawaii Int. Conf. Syst. Sci.*, vol. 3, 1998, pp. 317–326.

[39] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distrib. Comput. Column)*, vol. 32, no. 4 (Whole Number 121, December 2001), pp. 51–58, Dec. 2001, doi: 10.1145/568425.568433.

[40] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *Proc. 1st ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*2006, pp. 103–115.

[41] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proc. 28th ACM Symp. Princ. Distrib. Comput.*, 2009, pp. 312–313.

[42] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.

[43] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, Art. no. 11.

[44] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Trans. Comput. Syst.*, vol. 33, no. 2, pp. 1–32, 2015.

[45] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. Schultz, "Automatic reconfiguration for large-scale reliable storage systems," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 2, pp. 145–158, Mar./Apr. 2012.

[46] X. Liu et al., "ABSE: Adaptive baseline score-based election for leader-based BFT systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 8, pp. 1634–1650, Aug. 2025.

[47] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proc. Annu. Int. Cryptol. Conf.*, 2001, pp. 524–541.

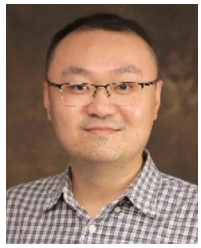
[48] X. Liu et al., "The deferred Byzantine generals problem," *IEEE Trans. Inf. Forensics Secur.*, vol. 20, pp. 7777–7792, 2025.

Xuyang Liu (Graduate Student Member, IEEE) received the BE degree in computer science and technology from the Beijing Institute of Technology, in 2022. He is currently working toward the PhD degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology, and also with the School of Computer Science, The University of Auckland. His research interests include applied cryptography, blockchain technology, and distributed consensus.

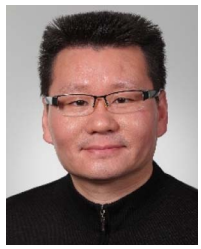
Zijian Zhang (Senior Member, IEEE) is currently a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is also a research fellow with the School of Computer Science, University of Auckland. In 2015, he was a visiting scholar with the Computer Science and Engineering Department, State University of New York at Buffalo. His research interests include design of authentication and key agreement protocol and analysis of entity behavior and preference.

Zhen Li is currently working toward the PhD degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include privacy computing, AI security, and blockchain.

Haibo Sun (Graduate Student Member, IEEE) received the BE degree in cyberspace security from Jilin University, in 2023. He is currently working toward the master's degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology. His research interests include applied cryptography, blockchain technology, and distributed consensus.



Meng Li (Senior Member, IEEE) received the PhD degree in computer science and technology from the School of Computer Science and Technology, Beijing Institute of Technology, China, in 2019. From 2017 to 2018, he was sponsored by China Scholarship Council (CSC), as a Joint PhD degree student supervised by Prof. Xiaodong Lin (IEEE fellow) with Broadband Communications Research (BBCR) Laboratory, University of Waterloo and Wilfrid Laurier University, Canada. From 2020 to 2021, he was sponsored by ERCIM 'Alain Bensoussan' Fellowship Programme, to conduct postdoc research supervised by Prof. Fabio Martinelli at CNR, Italy. From Mar. 2025 to Jun. 2025, he was supported by CSC as a Visiting Scholar collaborating with Prof. Mauro Conti (IEEE Fellow) with HIT Center, University of Padua, Italy. He was a postdoc researcher with the Department of Mathematics and HIT Center, University of Padua, Italy, where he was with Security and PRIvacy Through Zeal (SPRITZ) Research Group led by Prof. Mauro Conti (IEEE fellow). In Dec. 2025, he was promoted to a professor. He is currently a professor with the School of Computer Science and Information Engineering, Hefei University of Technology, China. He has authored or coauthored 135 papers in topmost journals and conferences, including *IEEE Transactions on Information Forensics and Security* (IEEE TIFS), *IEEE Transactions on Dependable and Secure Computing* (IEEE TDSC), *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Knowledge and Data Engineering*, TODS, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Services Computing*, *IEEE Communications Surveys and Tutorials*, IEEE S&P, USENIX Security, MobiCom, INFOCOM, and ISSTA. His research interests include security, privacy, applied cryptography, blockchain, TEE, and Internet of Vehicles. He was the recipient of 2024 IEEE HITC Award for Excellence (Early Career Researcher) and 2025 IEEE TCSVC Rising Star Award. He was selected into the IEEE Computer Society Computing's Top 30 Early Career Professionals for 2025. He is an associate editor for TIFS, TDSC, and *IEEE Transactions on Network and Service Management*. He was a TPC member for conferences, including ICDCS, Inscrypt, ICICS, and TrustCom. He is also a senior member of CIE, CIC, and CCF.



Jing Sun received the PhD degree in computer science from the National University of Singapore, in 2004. He is currently an associate professor with the School of Computer Science, University of Auckland, New Zealand. His research interests include AI-driven software engineering, secure and trustworthy software development, and the application of large language models to automated software analysis and verification. He has authored more than 140 research papers in leading international journals and conferences, including *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering and Methodology*, *Automated Software Engineering*, *ACM Computing Surveys*, and *Information Sciences*. His research interests include formal model repair, LLM-based code generation, and smart contract security analysis. He was the conference chair, program chair, and steering committee member for several top international conferences.

Jiamou Liu received the PhD degree in computer science from the University of Auckland, Auckland, New Zealand, in 2010. He is currently an associate professor with the School of Computer Science, University of Auckland. From 2011 to 2015, he was a senior Lecturer with the Auckland University of Technology, Auckland, and a researcher with the Department of Computer Science, Leipzig University, Leipzig, Germany, from 2009 to 2010. His research interests include social network analysis, multiagent systems, and algorithms.

Yong Liu is is currently a research fellow and doctoral supervisor of cybersecurity technology with Qi An Xin Technology Group Inc. His research interests include cloud computing security, vulnerability mining, and blockchain security evaluation.

Lei Xu received the PhD degree in electronic engineering from Tsinghua University (THU), Beijing, in 2015. From 2015 to 2017, she was a postdoctoral researcher with the Department of Computer Science and Technology, THU. She is currently an associate professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. Her research interests include differential privacy, federated learning, and applications of blockchain technology.

Jincheng An is currently an assistant research fellow with Qi An Xin Technology Group Inc. He is also an expert of cybersecurity and technology standardizing document drafting. His research interests include cyber resilience theory and evaluation, and data security detection.

Qi Sun is currently an algorithm researcher with Hangzhou Nuowei Information Technology Company Ltd., Hangzhou, China. Her research interests include NLP and privacy-preserving computing.

Liang Huang was born in 1984. He received the PhD degree. He is currently a senior engineer with Qi An Xin Technology Group Inc. His research interests mainly include cybersecurity, data security, and blockchain security.

Mauro Conti (Fellow, IEEE) received the PhD degree from the Sapienza University of Rome, Italy, in 2009. He was a postdoc researcher with Vrije Universiteit Amsterdam, The Netherlands. In 2011, he joined as assistant professor with the University of Padua, where he became associate professor in 2015, and full professor in 2018. He has been visiting researcher with GMU, UCLA, UCI, TU Darmstadt, UF, and FIU. He is currently full professor with the University of Padua, Italy. He is also affiliated with TU Delft and University of Washington, Seattle. His research ws funded by companies, including Cisco, Intel, and Huawei. He authored or coauthored more than 400 papers in topmost international peer-reviewed journals and conferences. His research mainly focuses on the area of security and privacy. He was the recipient of the Marie Curie Fellowship (2012) by the European Commission, and a Fellowship by the German DAAD (2013). He is also the editor-in-chief of *IEEE Transactions on Information Forensics and Security*, area editor-in-chief of *IEEE Communications Surveys & Tutorials*, and an associate editor for several journals, including *IEEE Communications Surveys & Tutorials*, *IEEE Transactions on Dependable and Secure Computing*, and *IEEE Transactions on Network and Service Management*. He was program chair for TRUST 2015, ICISS 2016, WiSec 2017, ACNS 2020, CANS 2021, and general chair for SecureComm 2012, SACMAT 2013, NSS 2021 and ACNS 2022. He is a senior member of the ACM and fellow of the Young Academy of Europe.

Liehuang Zhu (Senior Member, IEEE) is currently a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is also selected with the Program for New Century Excellent Talents in University, Ministry of Education, China. His research interests include cryptographic algorithms and secure protocols, Internet of Things security, cloud computing security, Big Data privacy, mobile and Internet security, and trusted computing.