Foundations of Dynamic BFT

Sisi Duan[†]
Tsinghua University
Email: duansisi@tsinghua.edu.cn

Haibin Zhang[*]
Beijing Institute of Technology
Email: haibin@bit.edu.cn

***Abstract*—This paper studies dynamic BFT, where replicas can join and leave the system dynamically, a primitive that is nowadays increasingly needed. We provide a formal treatment for dynamic BFT protocols, endowing them with a flexible syntax and various security definitions.**

We demonstrate the challenges of extending static BFT to dynamic BFT. Then we design and implement Dyno, a highly efficient dynamic BFT protocol under the partial synchrony model. We show that Dyno can seamlessly handle membership changes without incurring performance degradation.

I. Introduction

Byzantine fault-tolerant state machine replication (BFT) has been traditionally known as a primitive to build mission-critical systems. Nowadays, BFT has gained its prominence, because it is deemed as a core building block for blockchains: BFT is known as *the* model for permissioned blockchains [54], where the membership is static and the ledgers (replicas) know each other's identities but may not trust one another. BFT is also increasingly being used in various manners in permissionless blockchains (these protocols are also called hybrid blockchains [26, 27, 56]). This paper studies *BFT with dynamic membership*, or simply *dynamic BFT*, where replicas may join and leave the system dynamically. Dynamic BFT, a primitive that may be traditionally viewed as one enabling desirable features from the system perspective, is nowadays increasingly needed as a core building block for a myriad of blockchain and security applications.

Normal recovery and reconfiguration. As in any static distributed systems, a static BFT system has practical limitations [46]: if, for instance, one of the replicas crashes, needs maintenance, or is deemed as being faulty, the probability for the system to be always available reduces. It may not be always feasible to recover the faulty one, because recovering a node can take time, and more importantly, recovering may not always be possible (due to, e.g., permanent hardware failures). In this case, a better approach is to create a new replica to replace the faulty one.

Proactive recovery. Dynamic BFT can be used to build a robust BFT system with proactive recovery that works in the long run. The problem of proactively secure BFT systems has been studied in many previous works [10, 51]. These systems use a trusted hardware to periodically restart nodes, in a hope it will evict the adversary. The adversary (e.g., viruses), however, may well stay in the system, rendering the effort useless. A fresh replica, however, does not have such a problem. Dynamic BFT thus provides an alternative and arguably better solution to the traditional proactive recovery approach.

Consortium blockchains. The static membership property of BFT may significantly limit the applicability of consortium blockchains. The blockchain entities may choose (or be forced) to leave the constortium, while new entities may join the consortium. For instance, Libra, now rebranded as Diem, has already faced such a situation, where dynamic membership is managed by their smart contract after an agreement has been reached by the BFT.

Hybrid blockchains. The hybrid blockchains using BFT need to select a fixed number of BFT replicas, called committees, which can easily become sitting ducks. It is vital to be able to change committees while keeping the system up and running.

A. Technical Challenges and Our Contributions

A formal treatment of dynamic BFT. Despite the need for dynmaic BFT, there has, until now, been no rigorous formalization offered for dynamic BFT. While some related primitives have been defined in crash failure model [11, 46] and Byzantine failure model [20, 39, 40], there lacks a good abstraction for dynamic BFT with provable security. Looking at dynamic BFT from a modern vantage—in the "era of blockchains"—is long overdue. This paper fills in this gap and offers a formal treatment of dynamic BFT. Our specification covers a syntax separating BFT from the membership service, an approach stemming from the work of Schiper for the crash failure model [46]. Our definitions of security, however, take a rather different turn by treating indistinctively regular and membership requests. More crucially, our treatment has the following two features:

- In lifting security definitions to dynamic BFT, we define a new property (consistent delivery) that we find crucial to the security of dynamic BFT. We show some natural and alternative delivery properties fail to work.
- Our treatment consists of a number of security definitions for different notions of "being correct" for dynamic BFT, each being meaningful, from the weakest to the strongest we can envision.

Identifying (new) issues for dynamic BFT. Despite a long line of work for dynamic membership (mainly in crash fault-tolerant systems), we discover some (new) issues:

- *Problems due to dynamic quorum (specified in Sec. IV).* In dynamic BFT, the membership change leads to the the

[†]Sisi is with Institute for Advanced Study and BNRist.
[*]Corresponding author.

change of the quorum size, creating various liveness issues.

- *Problems due to view changes (specified in Sec. IV).* In dynamic BFT with view changes (i.e., leader election), there are various issues as well. For instance, the next leader may be unable to collect enough view change messages, may even be unaware of the view change occurring. Even worse, multiple replicas may claim they are the leader in the same view.
- *Problems due to message delivery (specified in Sec. III-E).* Static distributed systems have a message delivery assumption that messages transmitted among correct replicas are eventually delivered. This assumption does not hold for the dynamic setting, as some correct replicas, while being correct when sending messages, may leave the system in some future configuration. We find that even some classic protocols in the secure distributed computing community (e.g., [46]) simply assume message delivery across configurations and the proofs for these protocols are actually flawed.

Configuration discovery protocols. Our protocols rely on configuration discovery sub-protocols, where clients and new replicas obtain the membership of the system. The configuration discovery protocols are not just crucial from the functionality perspective but to the correctness of our dynamic BFT protocols. We provide one such protocol which uses explicit membership discovery. We also present in appendix two alternative protocols using implicit membership discovery.

Constructions, assumptions, and their presentation. We consider two different assumptions: the standard quorum assumption (assuming optimal resilience for each configuration) and the G-correct assumption (assuming a fraction of correct replicas never leave the system). Based on each assumption, we provide protocols satisfying different definitions of security we propose. We first present Dyno, the main protocol in the paper. We then present the variants of Dyno. We formally prove the security of all the protocols.

One of our aims is to handle membership change seamlessly. Dyno retains the full efficiency of the underlying BFT protocol and can handle membership requests independently of view changes. Dyno, however, does make significant modifications—while adding no performance overhead—to both the normal-case operation and the view change mechanism of the underlying BFT protocol. We show Dyno is highly efficient and robust under various join and leave scenarios.

II. Related Work

BFT. As a generic approach that tolerates arbitrary failures, BFT can be categorized into synchronous BFT [1], partially synchronous BFT [10, 14, 15, 50] and asynchronous BFT [16, 17, 33]. In this work, we focus on partially synchronous BFT.

Primary partition *vs.* partitionable membership services. A membership service may be primary partition or partitionable [11]. In a primary partition service, views at all replicas are totally ordered. In a partitionable one, views are partially ordered, i.e., multiple disjoint views may exist at the same time. The paper studies the primary partition model only.

Dynamic group communication. The group membership problem and the view synchronous communication problem were first discussed by Birman and Joseph [7]. A group membership abstraction provides a dynamic yet consistent view of active members. View synchronous communication [7, 47] extends group membership to support reliable broadcast within members of views. Extended virtual synchrony [37] extends virtual synchrony, ensuring a consistent relationship between delivery of messages and that of configuration changes across all replicas. Spread [5] and Secure Spread [4] encompass both virtual synchrony and extended virtual synchrony in the crash failure model. Secure Spread enhances Spread with authentication, integrity, access control, and confidentiality.

Chockler et al. [11] provide a comprehensive survey on dynamic group communication systems: the group membership service (for adding and removing processes) is defined first, while the communication primitives (e.g., reliable broadcast, atomic broadcast) are specified in a second step. In these systems, the group membership service is the basic layer of various communication stacks. In contrast, Schiper's specifications describe communication primitives first and then membership changes and allow that all membership changes come from *explicit* invocations of membership requests [46]. The primitive defined for atomic broadcast, for instance, is called dynamic atomic broadcast, or simply atomic multicast. Schiper argued that the choice leads to more "natural" and simpler specifications. Schiper [46] also shows that his specifications are only *slightly* different from the ones surveyed in [11] in terms of liveness properties.

Guerraoui et al. recently proposed dynamic Byzantine reliable broadcast (DBRB), where replicas can join and leave the system dynamically [20]. From the definition perspective, DBRB solves a rather different problem (broadcast) from ours (consensus) and focuses on asynchronous settings. From the technical perspective, DBRB allows divergent view paths that will eventually converge to the same view.

Rampart [39] and SecureRing [25] implement state machine replication in the Byzantine failure model and rely on Byzantine failure detectors to achieve liveness.

Reconfiguration for atomic storage. Dynamic atomic read/write (R/W) storage can be consensus-based [19, 36, 44] and consensus-free. Aguilera et al. demonstrate in DynaStore [2] that dynamic atomic storage can be solved, without using consensus (or randomization), in asynchronous settings. Starting from DynaStore, a number of consensus-free dynamic atomic storage constructions have been proposed (e.g., [3, 18, 23]). Kuznetsov and Tonkikh present asynchronous atomic storage with Byzantine failures [28].

Reconfiguration for SMR. Lamport proposes in Paxos to manage membership changes as part of the system state [30]. After an agreement is reached for a membership request, replicas wait for α batches of requests to be executed before they install the new configuration. Lorch et al. proposed SMART [34] where reconfiguration of the system is managed by creating an additional group of replicas. The two groups of replicas run parallel Paxos instances until the system state is fully migrated to the new group. For primary/backup replication, solutions for reconfiguration are studied in a crash failure setting [22, 49]. Raft [38] presents a two-phase approach for reconfiguration. The first phase is a transitional configuration called joint consensus. To transition to a new configuration, replicas in both configurations participate in the joint consensus, while agreement is independently maintained. This approach allows replicas in older configuration to continue providing service to the clients and replicas in new configuration to catch up with the history. Only after the joint consensus has been committed, the system moves to the new configuration. BFT-SMaRt is a BFT system that supports reconfiguration [50], where membership requests are treated as a special type of client requests and order all requests together. Our systems use this strategy as well. But doing so alone without further modifying the protocol, may create liveness issues (zero throughput), as we will theoretically show for any leader-based BFT protocols in Sec. IV and experimentally show for BFT-SMaRt in Sec. VIII.

Reconfiguration using auxiliary master. Reconfiguration of a system (storage, SMR, or primary backup replication) can be managed by an auxiliary master [31, 35, 43]. For instance, Vertical Paxos shows (in the crash failure setting) that it is sufficient to use $f+1$ replicas to build a configuration master that manages the configuration of SMR.

Membership management. Dynamic BFT is in sharp contrast to membership management, a service managing the nodes in distributed systems [12, 13, 24, 32, 41, 42, 45, 52]. The service can either be built from a standalone SMR (e.g, Apache Zookeeper [21], Google Chubby [9]) or via self discovery of the membership (e.g., SWIM [13]). In contrast, dynamic BFT is a fundamentally different primitive, which can be viewed as a self-configurable SMR.

Durability. Durability is a capacity of surviving state machine replication system crash or shutdown and bringing recovering replicas up to date. The property has been previously considered in the context of static groups [6, 10], while we consider a similar one for dynamic groups from a different perspective.

III. System and Threat Model

A. Static BFT

Conventional Byzantine fault-tolerant (BFT) protocols have a constant set of replicas, a fraction of which may fail arbitrarily (Byzantine failures). In BFT, a replica *delivers requests*, each submitted by a client. A request may include one or more operations. A replica executes the operations in the request and sends a reply to the corresponding client. The client computes a final response based on the reply messages.

In a system with n replicas, BFT tolerates $f \leq \lfloor \frac{n-1}{3} \rfloor$ Byzantine failures, which is optimal. Correctness for BFT is specified as follows:

- **Agreement**: If a correct replica delivers a request m, then every correct replica eventually delivers m.
- **Total order**: If a correct replica delivers a message m before delivering m', then another correct replica delivers a message m' only after it has delivered m.
- **Liveness**: If a correct client submits a request m, then a correct replica eventually delivers m.
- **Integrity**: No correct replica delivers the same message m more than once; if a correct replica delivers a message m and the client that submits m is correct, then m was previously submitted by the client.

Total order and integrity are safety properties, while the other two are liveness properties. Agreement and liveness together imply a client eventually receives a valid response. A BFT formalization may explicitly assign sequence numbers to client requests and ask correct replicas to execute requests according to the order.

Generally speaking, partially synchronous BFT protocols rely on view changes (with a form of leader rotations) for liveness and therefore proceed in views. A view change is triggered when a leader appears to be faulty (e.g., PBFT [10]), or triggered periodically according to some strategy (e.g., Spinning [53], HotStuff [55]). Asynchronous BFT protocols, however, do not have the view change mechanism.

B. Dynamic BFT

We consider a BFT system with replicas taken from a finite set $\Pi = \{p_1, p_2, \cdots\}$ (also called the universe). Each replica $p_i \in \Pi$ has a public/private key pair (pk_i, sk_i) and we assume for simplicity its public key is known by all processes in the set and serves as a unique identifier for p_i. A dynamic BFT group consists of a subset of Π. The replicas in a BFT group are *members* of the group. We use the notion of *configuration* to represent the successive membership of a BFT group. Let M_c be the membership (the group of replicas) of a configuration numbered by an integer c, initialized as 0 (the initial configuration). A replica changes its configuration via configuration installation. We introduce the following definitions:

Definition III.1. *A replica p is in configuration c, if $p \in M_c$.*

Definition III.2. *The (current) configuration of p is c if p has installed configuration c but has not installed another configuration after c.*

Definition III.3. *The latest configuration of the system is c, if at least one correct replica installs c and no correct replica has installed c' where $c' > c$.*

Definition III.4. *A replica p is* ***correct in configuration*** **c**, *if p installs configuration c, and p is correct (not faulty) in c. (Note p might or might not be faulty in configuration $c+1$, but is not faulty in c.)*

Definition III.5. *A replica p is* **c-correct**, *if 1) p is correct in configuration c, and 2) no correct replicas in c install any configuration greater than c, or some correct replica in c installs configuration $c+1$ after c and $p \in M_{c+1}$. (p is c-correct if c is the latest configuration for all correct replicas in c, or p is in the next configuration of c.)*

Definition III.6. *A replica p is* ***c-faulty*** *if $p \in M_c$ and p is not c-correct. (A replica p can be correct in configuration c but* *c-faulty, as, for instance, p may be removed from the group precisely after configuration c but before configuration $c+1$ and is never in configuration $c+1$.)*

Definition III.7. *A replica p is* **g-correct***, if 1) p is 0-correct (where 0 is the initial configuration of the system), and 2) there does not exist a $c > 0$ such that p is not c-correct. (A g-correct replica is correct in all its configurations.)*

Definition III.8. *A replica p is* **g_c-correct** *for $c \geq 0$, if 1) p is c-correct, and 2) there does not exist a c' such that $c' > c$ and p is not c'-correct. (A g_c-correct replica is correct in configurations $c' \geq c$. A g_0-correct replica is g-correct.)*

Our notion of configurations (dealing with membership) is independent of views (in the sense of view changes for partially synchronous BFT). A dynamic partially synchronous BFT can have both the notion of configurations and the notion of views. Note a static BFT group is a special case of the dynamic BFT model with a fixed number of replicas and a single configuration.

C. Group Membership Change for Dynamic BFT

We consider two *membership requests* that can modify the membership: the *join* request that adds a new replica to the group and the *leave* request that removes a replica from the group. The membership requests are different from *regular* BFT requests (write and read requests) that do not change the membership. We assume without loss of generality that membership requests are the only method of modifying the membership. We levy no restrictions on who may issue membership requests. They may be invoked by any *authorized* clients (e.g., replicas joining or leaving the system, system administrators, trusted CAs, high-level programs). We also do not discuss *why* a replica is included or excluded but only present *how* we do so. In practice, as illustrated in the examples in Sec. I, there is typically no ambiguity in agreeing on who will join or leave the group and why and when the group membership needs to change.

Upon execution of a join request for p_i at a replica p_j in configuration c, the membership for p_j becomes $M_c \cup \{p_i\}$ (and correspondingly p_j installs the new configuration). Upon execution of a leave request for p_i at a replica p_j in configuration c, its membership becomes $M_c - \{p_i\}$.

For two configuration memberships M_i and M_j with $i < j$, we use M_j/M_i to represent the replicas that exist in M_j but do not exist in M_i. For instance, if $M_i = \{p_2, p_3, p_4\}$ and $M_j = \{p_3, p_4, p_5, p_6\}$, then $M_j/M_i = \{p_5, p_6\}$. Let n_c and t_c be the number of replicas and c-faulty replicas in configuration c, respectively. In configuration c, we have $n_c = |M_c|$.

D. Assumptions

We now specify the system assumptions.

Standard quorum assumption. We assume the optimal resilience model in this work. Namely, the maximum number of failures the system can tolerate in configuration c is $f_c \leq \lfloor \frac{n_c-1}{3} \rfloor$. A *quorum* of replicas consists of at least $\lceil \frac{n_c+f_c+1}{2} \rceil$ replicas. We use Q_c to represent the quorum size in configuration c. This is a standard and optimal assumption. Unless mentioned otherwise, we consider the default assumption.

G-correct assumption. We also consider a stronger assumption which is not essential but may yield some (much) simpler protocols for some cases. Let $F = max(\{f_c\}) + 1$ for all $c \geq 0$. G-correct assumption requires there exist at least F replicas that are correct across all configurations. In other words, "enough" correct replicas never leave the system.

In addition, we make two standard assumptions for dynamic membership (used in all other such scenarios): first, the number of replicas that join or leave the system is bounded (a natural assumption is that from configuration c to $c+1$, at least Q_c c-correct replicas are still in $c+1$); second, the initial configuration is known by all replicas in the universe Π.

E. Property Specification

In our formalization, membership requests and regular BFT requests may be collectively called requests. We treat membership requests in the same way as regular requests. Thus, when specifying the correctness of dynamic BFT below, a request m may be either a membership request or a regular request. In particular, an invocation of a join or leave request is just an invocation of a special client request. An execution of a join (resp. leave) request corresponds to the delivery of the request and installation of a new configuration via the *add* (resp. *remove*) operation in the request.

- **Agreement**: If a correct replica in configuration c delivers a request m, then every correct replica in configuration c eventually delivers m.
- **Same configuration delivery**: If a correct replica p_i (resp. p_j) delivers m in configuration c^i (resp. c^j), then $c^i = c^j$.
- **Total order**: If a correct replica in configuration c delivers a message m before delivering m', then another correct replica in configuration c delivers a message m' only after it has delivered m.
- **Liveness**: If a correct client submits a request m, then eventually a correct replica in some configuration c delivers m.
- **Integrity**: No correct replica delivers the same message m more than once; if a correct replica delivers a message m in some configuration and the client that submits m is correct, then m was previously submitted by the client.

The same configuration delivery property is a natural requirement for the membership service and is defined previously (e.g., [11, 46]). A BFT system may assign increasing sequence numbers to different client requests to order transactions. Under such a syntax, we may unify the same configuration delivery property and total order property:

- **Enhanced total order**: If a correct replica in configuration c delivers a request m with a sequence number, and another correct replica in configuration c' delivers a request m' with the same sequence number, then $m = m'$ and $c = c'$.

A practical, durable BFT system relies on a state transfer protocol to bring recovering replicas or replicas that have fallen behind its peers up to date [6, 10]. In dynamic BFT, new replicas joining the group also need the state transfer mechanism to catch up with other replicas. New replicas or recovering replicas may be deemed as being faulty before they obtain the system state.

Some, but not all, existing BFT protocols are specified without explicit agreement property, but these protocols can easily satisfy it. This is, however, not the case for dynamic BFT, as achieving agreement is not at all trivial; hence, we explicitly consider agreement in all our dynamic BFT constructions.

The need for additional properties. Having discussed dynamic BFT properties naturally mirroring properties from static BFT, let us now motivate the need of additional properties. In static BFT settings, agreement, total order, and our liveness definition imply that a correct client will receive replies to its requests. More concretely, for a client request m, agreement and liveness guarantee that all correct replicas will eventually deliver m. Since there are $n \geq 3f + 1$ replicas, a client can safely deliver a response through the majority voting approach, computing a response after receiving $f+1$ matching replies. In fact, a typical definition of liveness in static BFT already requires that a client will eventually receive replies to its request so the agreement property is not "needed".

The classic definitions do not immediately work for dynamic groups. First, not knowing the membership of the system or the total number of replicas in some configuration, the client may never be certain when to deliver the replies to the requests. Second, even if the client knows the membership information, the membership may have changed by the time a message is processed. These two factors motivate the need of defining additional properties.

Attempt-1. A "natural" enhancement to liveness that could circumvent the membership issue may be defined as follows:

- If a correct client submits a request m, then the client eventually delivers a response from replicas.

Unfortunately, the definition does not capture any meaningful correctness guarantees in the dynamic BFT setting. One may easily design a trivial protocol satisfying the above definition, where a correct client receives an irrelevant response.

Attempt-2. Another choice is to define the following property:

- **Membership discovery**: A correct client having submitted m can eventually learn about the membership of the configurations for which m is delivered.

The membership discovery property requires the client eventually knows the membership information (the identities of replicas) for the configuration *where its message is delivered*. It does not say if a client can or need to learn the *latest* (current) membership, which is challenging in asynchronous environments (even if we assume a partially synchronous model for liveness). Indeed, even if a client obtains the latest membership information, the membership may have been changed when the client submits the request or when the request is being processed. The membership discovery property appears general to encompass many scenarios we could envision, whether the client obtains the membership information via a standalone subprotocol (either before or after its requests are processed), or obtains the membership information during the protocol runtime.

Intuitively, the membership discovery property, together with agreement, liveness, and total order, would imply that a client knows when it receives enough matching replies from the same configuration and can safely deliver a response based on majority voting. More precisely, with a client request m, agreement and liveness guarantee that all correct replicas in some configuration will eventually deliver a client message m; total order guarantees that m has to be delivered within the same configuration, say, c; knowing the precise membership of the configuration c that delivers m and the total number of members in the configuration c, a client can safely deliver a response through the conventional majority voting approach, computing a response after receiving $f_c + 1$ matching replies in the configuration c.

Unfortunately, the above argument has a major flaw. Even if the client learns the information of the configuration c for which the request m is delivered and even if we assume perfect channels, the reply messages sent by correct replicas in configuration c may not be able to reach the client. This is because the perfect channel guarantees message delivery only when the sender is—all the time—correct, but the sender may leave the system in some future configurations; indeed, due to network asynchrony, a reply message from some replica p_i may not reach the client before p_i leaves the system.

We comment that while we do not adopt the definition of membership discovery, our constructions are indeed motivated by the idea.

Final attempt. Finally, we define the following property:

- **Consistent delivery**: A correct client submitting m will deliver a correct response which is consistent with the state in some configuration where m is delivered.

The property is slightly unconventional, encompassing both liveness and safety. One can, however, easily divide the property into a liveness one (a correct client eventually receives a reply to its request) and a safety one (the delivered request by a correct client is consistent with the state of some configuration where the request is delivered).

To formally define consistent delivery, we use the *state machine replication* notation [29, 48]. A deterministic state machine consists of the following tuple of values: a set of states, a set of requests, a set of replies, a transition function that takes as input a request and a state and outputs an updated state, and an output function that takes as input a request and a state and outputs a reply. Requests are submitted by clients and replies are sent to clients. The states encode the system state (including configurations and various variables kept).

We say a reply *re* is consistent with a state *st* for a correct state machine, if the reply *re* was generated by the output function for some request and the state *st*. As we consider deterministic state machines, and as total order is achieved by all correct replicas, the same request and the same state determines the same reply for correct replicas.

	if clause	main clause	assumption	constructions
V	correct in c	g_c-correct	standard quorum	Dyno-S
				Dyno
V_1	correct in c	c-correct	G-correct	Dyno
			standard quorum	Dyno-A
V_2	correct in c	correct in c	G-correct	Dyno-C
			standard quorum	Dyno-AC
V'	c-correct	g_c-correct	same as V	
V_1'	c-correct	c-correct	same as V_1	
V_2'	c-correct	correct in c	same as V_2	

TABLE I: Overview of agreement definitions and constructions. The definition variants can also be applied to other dynamic BFT properties, but they do not (seem to) lead to interesting efficiency trade-offs.

Note that correct replicas may generate a reply (and a corresponding state) in some configuration, but as some replies may never reach the client in this configuration, the client will not deliver the replies in this configuration. (Recall that correct replicas may have left the system before the reply can reach the client.) We, however, do want to ensure that the client will eventually deliver a reply that is consistent with the state that the system maintains.

Also note in a conventional, static state machine replication setting, a reply is always delivered in the same configuration, as there are enough correct replicas. In this setting, one does not have to define this additional "consistent delivery."

F. Variants of Definitions

There are a number of variants for the definitions in Sec. III-E. For instance, for agreement, both if-clauses and main-clauses discuss "correct replicas in c." We summarize in Table I these variants, with definitions described above in detail being V_2, the strongest one we can envision. We illustrate their differences using the agreement property[1], and different agreement properties lead to different constructions with interesting efficiency trade-offs.

One can similarly describe definitions alike for other properties. The definition variants for these properties do not (seem to) lead to more efficient constructions. So this paper only provide constructions with different agreement definitions.

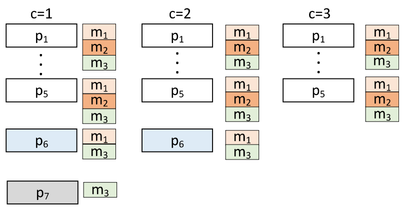


Fig. 1: Illustration for variants of definitions.

We use Fig. 1 to explain the agreement definitions. In the example, there are three configurations and seven replicas. Replicas p_1 to p_5 are correct and g_1-correct. Replica p_6 is 1-correct but 2-faulty as it leaves the system after configuration 2. Replica p_7 is correct in 1 but 1-faulty as it leaves the system after configuration 1. For agreement definition V, if the correct replica p_1 in configuration 1 delivers a message m_2, any other g_1-correct replicas, including p_2 to p_5, will eventually deliver m_2. Replicas p_6 and p_7, however, may or may not deliver m_2. For agreement definition V_1, if a 1-correct replica p_1 delivers m_1, any other 1-correct replicas, including p_2 to p_6, have to deliver m_1. For V_2, an example is m_3. In particular, if a correct replica in configuration 1 p_1 delivers m_3, any other correct replicas in configuration 1, including p_2 to p_7, will deliver m_3. The properties for V', V_1', and V_2' are similar, except that the if clause requires c-correct replicas.

Which of the six versions should one choose? The if-clauses in each version represents the conditions of the properties: if weaker conditions are used, we will have stronger properties. Note that V (resp. V') is weaker than V_1 (resp. V_1') and V_2 (resp. V_2'), as g_c-correct replicas are also c-correct replicas and correct replicas. Furthermore, V_1 (resp. V_1') is weaker than V_2 (resp. V_2') as the set of correct replicas in some configuration c includes the set of c-correct replicas. One could compare our setting with that of uniform vs. non-uniform broadcast primitives for crash failures, where uniform broadcast primitives are stronger ones.

G. Comparison with Previous Specifications

Our syntax follows that of Schiper to separate dynamic BFT from its membership service [46]. As argued by Schiper, the treatment is more natural and simpler than those of Chockler et al. [46]. Our specification for security (correctness) properties, however, is significantly different from that of Schiper [46]. First, our specification deals with Byzantine failures, while that of Schiper is designed for the uniform broadcast primitives in the crash failure model.

Second, while we define properties for regular requests and membership requests all together, Schiper defined all the properties for regular requests and membership requests separately. In particular, Schiper's work describes first *total order for regular requests* and then *total order for membership requests*, both in the conventional sense of request total ordering. The total order specification of Schiper alone may cause the following "anomaly," as shown in Fig. 2: the correct replica p_1 may deliver a membership request before a regular request, while the correct replica p_2 delivers the regular request before the membership request. This does not violate the two separate total order properties defined in Schiper's work, but does violate the total order guarantee we define.

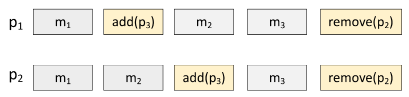


Fig. 2: An example that does not violate the total order property in Schiper's scheme but violates our definition.

That is, the two total order properties defined in Schiper's work, even when combined, is strictly weaker than ours. With the "same configuration delivery" properties defined additionally in Schiper's work for both regular requests and mem-

[1]Strictly speaking, we could have three more variants by asking the replicas in the if clauses to be "g_c correct." These definitions are weaker and do not seem to lead to interesting (more efficient) constructions.

bership requests, however, one can check that the "anomaly" would not occur. In fact, the two total order properties and the two same configuration delivery properties defined in Schiper's work, all together, are equivalent to our enhanced total order property (which encompasses the same configuration delivery property). Therefore, while there is no one-to-one correspondence between our properties and those of Schiper, our specification and the specification of Schiper (with appropriate modifications for Byzantine failures) can be *as a whole* equivalent. Our definitions, however, appear cleaner and simpler.

Last, Schiper's work does not define the needed property (consistent delivery) that is crucial to the security of dynamic BFT.

IV. Challenges of Dynamic BFT

When building a dynamic BFT, one may simply treat membership requests as regular requests. The intuitive approach, however, leads to various issues, making dynamic BFT fail to achieve liveness. We illustrate below some challenges for leader-based BFT protocols with view changes in partially synchronous environments.

Liveness problems in normal-case operations. The agreement property, regardless of the partially synchronous or asynchronous assumption, requires that if a correct sender stays online for a sufficiently long time, a correct receiver will receive the messages from the sender. This, however, is not the case for dynamic BFT. In particular, consider a c_q-correct replica p_i that leaves the system immediately after c_r where $c_r > c_q$. Replica p_i has delivered m in c_q and remove(p_i) in c_r. Consider another correct or c_q-correct replica p_j which is still in c_q. The replica needs to collect Q_{c_q} matching messages to deliver m. Some c_q-correct replicas (or even all c_q-correct replicas besides p_j), however, might not be correct any more as they already move to c_r, e.g., p_i has already left the system. Therefore, correct replicas might not be able to deliver m, creating agreement and liveness problems.

Liveness problems for view changes. We summarize various liveness problems associated with view changes.

1) The *correct* leader in the new view may not be able to obtain enough view change messages to start a new view.
2) The designated leader in the new configuration may not be aware of the view change. This is because replicas stay in different configurations and they fail to send all replicas view change messages in the latest configuration.
3) Multiple replicas may claim that they are the new leader after the view change.

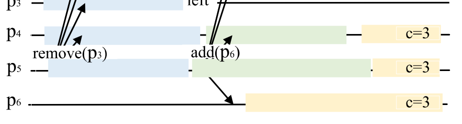


Fig. 3: The liveness challenge for view changes.

We show an example in Fig. 3. Replicas move from configuration $c = 1$ to $c = 3$, where in all three configurations, p_5 is the leader. In configuration $c = 1$, there are five replicas and p_3 is removed. Replicas p_2, p_4, p_5 install configuration 2 while p_1 still stays in $c = 1$. In configuration 2, quorum size is 3 so p_2, p_4, and p_5 are able to reach an agreement. In $c = 2$, p_6 requests to join the system. After the delivery of the request, replicas p_4, p_5, and p_6 install $c = 3$ while p_2 still stays in $c = 2$. If view change occurs at this stage, replica p_6 is the designated leader in the new view. Since quorum size is 4 in $c = 3$, p_6 has to collect 4 messages to start a new view. However, p_1 and p_2 are not even aware of p_6. Furthermore, replica p_1 may believe it is the new leader. Therefore, all correct replicas will halt forever since replicas do not process any messages during view changes and replicas may never resume to normal-case operation. In a static system, the problem could be fixed by having replicas synchronize with other replicas. In a dynamic system, however, replicas may not even know the identities of replicas currently in the system.

V. Overview of Our Protocols

We provide secure constructions satisfying various dynamic BFT definitions in Table I based on two assumptions in partially synchronous environments.

A. Dyno-S: An Intuitive but Inefficient Construction

Dyno-S formalizes the intuition that each time a membership request is processed, a view change needs to be triggered to prevent replicas from missing newer configurations. However, constructing such a protocol—with provable security—using the idea is not trivial. The approach needs to solve the challenges in Sec. IV and needs to achieve consistent delivery. Due to space restrictions, we will present the construction in our full paper. The major drawback of the construction is that every view change results in a window of zero throughput, as replicas do not process any requests during view chagnes.

B. Dyno, Dyno-C, and Dyno-AC

Dyno is our main protocol that retains its efficiency during membership changes. It is secure in the sense of definition V under the standard quorum assumption and is also secure in the sense of V_1 under the stronger G-correct assumption. Jumping ahead, we also provide dynamic BFT constructions with stronger properties (summarized in Table I). Dyno-A is built on top of Dyno and additionally has a terse addendum (A) protocol running. Dyno-C differs Dyno in a minor manner, using a slightly constraint (C). Dyno-AC, as its name suggests, incorporates the addendum protocol (A) and the modified constraint (C). We will describe what the addendum protocol and the constraint mean in Sec. VII.

Below, we describe Dyno at a high level. We define two events when describing our membership approach: *init*() and *deliver*(). The *init*() event is triggered where the primary in the current view starts a new round of the protocol. The *deliver*() event denotes the event that a request is committed and ready to be delivered during normal-case operation. Our membership protocol has the following key components to ensure correctness without performance interruption.

We manage configuration as part of the system state, similar to prior works (e.g., Paxos, Schiper's work, and BFT-SMaRt). The membership request is treated in the same way as regular requests. Replicas reach agreement on a batch of requests in each round. If the batch consists of both regular requests and membership requests, the regular requests are delivered first according to a deterministic order (same as those in BFT with static membership). After that, membership requests are delivered. All replicas then install the new configuration.

Temporary membership management. When a join membership request is delivered, the configuration might be different from that when the request is submitted. To simplify the notification process, we introduce the concept of *temporary membership*. Specifically, each new replica acts as a *learner*. Existing replicas send the normal-case operation messages to both replicas in the current configuration and all the temporary members. The quorum size, however, still remains the same as the current configuration. Upon the *deliver*() event, the membership is updated and becomes the same as temporary membership. Doing so allows us to have a more efficient state transfer mechanism.

Configuration discovery protocols and consistent delivery. We devise configuration discovery protocols in order for new replicas and clients to learn the membership of the current system. We provide protocols achieving the goal implicitly and explicitly. The protocols are also vital to achieve consistent delivery, as the client needs configuration discovery to verify replies from the replicas currently in the system.

There are two more techniques we need to achieve consistent delivery. First, we enforce agreement across configurations to ensure all correct replicas, not just a fraction of correct replicas, up to date. Second, replicas maintain a total order of execution history and configuration history (a subset of execution history). The execution history contains proof of delivery (in the form of signatures, proving that a quorum of replicas have agreed to deliver the request) for each request to ensure total order and agreement. The configuration history includes the proof of delivery for membership requests only. The configuration history is used to verify if the correct replica has installed certain configurations. The configuration history allows a client to determine if it has received enough matching replica replies to safely deliver them in some configuration. Note it is possible the client cannot deliver the replies in the configuration where the request is issued, but the client will deliver them in some future configuration.

Carefully designed view changes. To ensure that the designated leader in a new view receives a quorum of view change messages, we introduce additional workflow during view changes. Specifically, each replica includes its configuration number in a view change message together with a valid proof of delivery for the membership requests.

For any replica p_i in configuration c, it compares c to c', where c' is the configuration number carried in a view change message sent by p_j. If c' is smaller, p_i forwards the view change message to all replicas in $M_c/M_{c'}$. Note that $M_c/M_{c'}$ is sufficient as the replica p_j (if correct) must have already sent the message to $M_{c'}$. On the other hand, if c is smaller, there are two cases: p_i has previously sent a view change message; p_i has not previously sent a view change message. In the former case, p_i re-sends its view change message to $M_{c'}/M_c$. In the latter case, p_i will send its view change message (if applicable) directly to all replicas in $M_{c'}$. This ensures that replicas in the new view is able to collect a sufficiently large number of view change messages to enter the new view.

VI. The Dyno Protocol

We now present Dyno, focusing on how membership requests and regular requests are processed and the workflow of view changes. Throughout the paper, we assume each message m is unique. In this section, we use the normal-case operation as an oracle, denoted by the *init*() and *deliver*() events, as discussed in Sec. V. In particular, we use Bracha's reliable broadcast [8] as the normal-case operation and we discuss the details of how the *init*() and *deliver*() events are triggered in Appendix A. We use Bracha's broadcast since it is a primitive that achieves agreement, which we find crucial for dynamic BFT. For each request, we use *proof of delivery* to denote the proof that the request can be safely delivered. In our case, this refers to $2f + 1$ matching messages in the second phase of Bracha's broadcast (or prepare certificate in the PBFT term) or $f + 1$ matching messages in the third phase (or $f + 1$ $\langle \text{COMMIT} \rangle$ messages in the PBFT term).

When we describe the protocol, we consider a configuration discovery protocol triggered via the $ObtainConfig()$ function. Replicas and clients can query the function to obtain the current configuration. The detail of configuration discovery is discussed in Sec. VI-B. In this section, we ignore the details of the garbage collection as conventional checkpoint protocols can be used in our system.

A. The Protocol

Clients submitting regular requests. A client is able to obtain a configuration c and the set of replicas M_c via the $ObtainConfig()$ function. To submit a request, a client first broadcasts a message $\langle \text{SUBMIT}, c, \langle \text{REQUEST}, cid, o\rangle\rangle$ to all replicas in configuration c. The request $\langle \text{REQUEST}, cid, o\rangle$ is a regular request with a valid signature, where cid is the id of the client and o is the operation. There are two cases for the replies the client might get. 1) If the client gets a reply from $f_c + 1$ replicas in M_c, it completes the request. 2) The client may get a reply from a replica in c' where $c' > c$. In this case, the client verifies the configuration history $chist$ (described in great detail later) and checks whether c' is a valid configuration. If c' is valid, the request is completed. If none of the cases apply and the client times out, the client performs configuration discovery again and submits the request, until the request is completed.

Normal-case operation and membership requests. The pseudocode for normal-case operation at replica p_i is shown in Fig. 5. When a replica in c receives a request from a client

```
Initialization
  c                                                    {current configuration}
func submit()
  ▷ to join                                            {new replica}
    $c, M_c \leftarrow ObtainConfig()$
    broadcast $\langle$SUBMIT$, c, \langle$JOIN$, pk\rangle\rangle$ to $M_c$
    start timer $\Delta$
    upon receiving $\langle$CONF$, c', M_{c'}, chist'\rangle$   {configuration notification}
      if $chist'$ is valid and $c' > c$
        $c \leftarrow c'$, $M_c \leftarrow M_{c'}$
    upon receiving $2f_c + 1$ $\langle$HISTORY$, s, h, \mathcal{C}, \mathcal{P}\rangle$
      for $m$ in $h$, deliver($m$)
    upon deliver(batch) where $\langle$ADD$, i, m\rangle \in batch$
      wait until state transfer is completed
      stop $\Delta$, complete the request
  ▷ to leave                                           {existing replica}
    broadcast $\langle$SUBMIT$, c, \langle$LEAVE$, i\rangle\rangle$ to $M_c$
    start timer $\Delta$
    upon deliver(batch) where $\langle$REMOVE$, i, m\rangle \in batch$
      stop $\Delta$, complete the request
  ▷ to submit a regular request                        {client}
    $c, M_c \leftarrow ObtainConfig()$
    broadcast $\langle$SUBMIT$, c, \langle$REQUEST$, cid, o\rangle\rangle$ to $M_c$
    start timer $\Delta$
    upon $f_c + 1$ matching $\langle$REPLY$, c, re\rangle$ from $M_c$
      stop $\Delta$, complete the requst
    upon $f_{c'} + 1$ matching $\langle$REPLY$, c', re, chist'\rangle$ from $p_j \in M_{c'}$
      stop $\Delta$, complete the request
upon timeout($\Delta$)
  repeat submit()
```

Fig. 4: Pseudocode for a client/a replica p_i that issues requests.

that has c' in the $\langle$SUBMIT$\rangle$ message, there are two conditions. 1) If the request has already been delivered, p_i directly replies with $\langle$REPLY$, c', re, chist\rangle$, where re is the execution result and $chist$ is the configuration history. 2) Otherwise, the request is added to the pending queue. Meanwhile, if $c = c'$, the replica forwards the request to the leader in the current configuration. If $c > c'$, the client is still in a prior configuration, p_i forwards the request to $M_c/M_{c'}$.

For a replica to join the system, the replica broadcasts a $\langle$SUBMIT$, c, \langle$JOIN$, pk\rangle\rangle$ request, where pk is the public key. To remove a replica from the system, the replica or the administrator broadcasts a $\langle$SUBMIT$, c, \langle$LEAVE$, i\rangle\rangle$ message.

When the current leader of the system has a non-empty queue, the leader obtains a batch of request(s) from the pending queue. If there is a $\langle$JOIN$, pk\rangle$ request, the leader adds a $\langle$ADD$, j, m\rangle$ message to the batch, where m is a valid $\langle$JOIN$\rangle$ request and j is the identifier assigned to the replica. If there is a $\langle$LEAVE$, j\rangle$ request, the leader adds a $\langle$REMOVE$, j, m\rangle$ message to the batch, where m is a valid $\langle$LEAVE$\rangle$ request. Then the leader triggers the *init*() event. If a non-primary replica triggers the *init*(*batch*) event for a *batch*, for each $\langle$ADD$, j, m\rangle$ in the batch, the replica adds p_j to TM and continues the normal-case operation where it also sends the messages to p_j. The value of f and quorum size, however, are still determined by the group members M. In other words, $f_c = \lfloor \frac{|M|-1}{3} \rfloor$ and the quorum size is $\lceil \frac{|M|+f_c+1}{2} \rceil$.

For any replica p_j that requests to join the system, if a replica p_i maintains a higher configuration number, it executes the $sync()$ function and sends the configuration history to p_j. This is used for p_j to directly obtain the latest configuration of the system.

```
Initialization
  $c, M, TM$        {configuration, membership, temporary membership}
  $chist$                                    {confiugration history}
▷ events
  upon receiving $m = \langle$SUBMIT$, c', \langle$REQUEST$, cid, o\rangle\rangle$
    if $m$ has been delivered
      send $\langle$REPLY$, c, re, chist\rangle$ to client $cid$
    else
      if $c = c'$, forward $m$ to leader
      else if $c > c'$, forward $m$ to $M_c/M_{c'}$
      add $m$ to queue
  as a leader
    upon non-empty queue
      $batch \leftarrow$ regular requsts in queue
      for each $m = \langle$SUBMIT$, c', \langle$JOIN$, pk\rangle\rangle$ in the queue
        $j \leftarrow AssignID()$
        $batch \leftarrow batch \cup \langle$ADD$, j, m\rangle$
      for each $m = \langle$SUBMIT$, c', \langle$LEAVE$, j\rangle\rangle$ in the queue
        $batch \leftarrow batch \cup \langle$REMOVE$, j, m\rangle$
      init(batch)
  as a replica
    upon init(batch)
      if $\langle$ADD$, j, m\rangle \in batch$
        $TM \leftarrow TM \cup \{p_j\}$
        $sync(m)$
▷ utility functions
  func $sync(\langle$ADD$, j, m\rangle)$ where $m = \langle$SUBMIT$, c', \langle$JOIN$, pk\rangle\rangle$
    if $c > c'$
      send $\langle$CONF$, c, M_c, chist\rangle$ to $p_j$
  func deliver(batch)
    for $m = \langle$SUBMIT$, c', \langle$REQUEST$, cid, o\rangle\rangle$ in batch
      $reply(m)$
    if there are membership requests
      $c \leftarrow c + 1$
      for $\langle$ADD$, j, m\rangle$ in batch
        $M \leftarrow M \cup p_j$
      for $\langle$REMOVE$, j, m\rangle$ in batch
        $M \leftarrow M - p_j$
      send $\langle$HISTORY$, s, h, \mathcal{C}, \mathcal{P}\rangle$ to $M_c/M_{c-1}$
      $chist \leftarrow chist \cup batch$
  func $reply(\langle$SUBMIT$, c', \langle$REQUEST$, cid, o\rangle\rangle)$
    if $c > c'$, send $\langle$REPLY$, c, re, chist\rangle$ to $cid$
    else, send $\langle$REPLY$, c, re\rangle$ to $cid$
```

Fig. 5: Normal-case operation at replica p_i.

```
Initialization
  $v, c$                                       {view, configuration}
as a replica
  func view-change()
    $v \leftarrow v + 1$
    broadcast $m' = \langle$VIEW-CHANGE$, v, c, \mathcal{C}, \mathcal{P}, \mathcal{PP}, i\rangle$
  upon receiving $m = \langle$VIEW-CHANGE$, v, c', \mathcal{C}, \mathcal{P}, \mathcal{PP}, j\rangle$
    if $c' < c$
      send $m$ to $M_c/M_{c'}$
    if $c' > c$ and $Verify(\mathcal{PP}, \mathcal{P})$
      $M_c \leftarrow M_{c'}$
      send $\langle$VIEW-CHANGE$, v, c, \mathcal{C}, \mathcal{P}, \mathcal{PP}, i\rangle$ to $M_{c'}/M_c$
as the new leader
  upon receiving $Q_c$ $\langle$VIEW-CHANGE$, v, c', \mathcal{C}, \mathcal{P}, j\rangle$
    broadcast $\langle$NEW-VIEW$, v, c, \mathcal{V}, \mathcal{O}\rangle$
```

Fig. 6: View change protocol at replica p_i.

Upon the *deliver*() event, replica p_i first delivers regular requests and then the membership requests. If membership requests are included in the batch, p_i installs a new configuration and increases c by 1. For each $\langle$ADD$, j, m\rangle$ message, p_i adds p_j to M, and starts state transfer to p_j. For each $\langle$REMOVE$, j, m\rangle$ message, p_i removes p_j from M. If p_j delivers the REMOVE request, it leaves the system.

Any replica p_i that requests to join the system acts as a *learner* that passively learns the results from the normal-case operation, i.e., it is added to the temporary members by replicas in the system. Learners process the normal-case messages following the same rules as existing replicas.After p_i delivers the join request, it waits until it completes state transfer by accepting $2f_c + 1$ $\langle \textsc{History}, s, h, \mathcal{C}, \mathcal{P}\rangle$ messages. After that, p_i participates in the normal-case operation. The state transfer for a new replica to catch up with the execution history and the maintain proof of delivery for historic requests.

View change. We now sketch the major workflow for view changes, the pseudocode of which is shown in Fig. 6.To start view change, each replica p_i increments its view number and broadcasts a $\langle \textsc{View-change}, v, c, s, \mathcal{C}, \mathcal{PP}, i\rangle$, where v is the view number, c is the configuration number, $\mathcal{C}$ is a stable checkpoint, $\mathcal{P}$ is a set of proofs of delivery for requests since last stable checkpoint, $\mathcal{PP}$ is a set of membership requests.

Upon receiving a View-change message from replica p_j, a replica compares c' carried in the message with its local configuration number c. If $c' < c$, it means that the replica p_j has not installed newer configuration(s) so some replicas in M_c cannot receive the message. In this case, p_i forwards the View-change message to M_c. If $c' > c$, it means that replica p_i has not installed newer configurations. In this case, p_i broadcasts a View-change message to $M_{c'}$. $M_{c'}$ can be obtained from $\mathcal{PP}$ and the proof of delivery can be obtained from $\mathcal{P}$.

Consider that c is the configuration installed by at least one correct replica and no correct replica has installed a configuration greater than c, the new leader can be identified by $M_c[v \bmod |M_c|]$. When the designated leader in the new view collects $2f_c + 1$ valid View-change messages, it enters the new view by broadcasting a $\langle \textsc{New-view}, v, c, \mathcal{V}, \mathcal{O}\rangle$, where $\mathcal{V}$ is a set of View-change messages and $\mathcal{O}$ is a set of normal-case operation messages. Replicas then resume normal-case operation and process messages in $\mathcal{O}$ accordingly.

B. Configuration Discovery

Configuration discovery ensures that new replicas and clients learn the membership of existing system. The approach we present in Dyno requires clients and new replicas to discover newer configurations (if any). In this section, we first define configuration history. Then we present the self-discovery approach we use in our protocol by defining the interface used by the normal-case operation. We describe two alternative constructions in Appendix B. The proofs for all the configuration discovery protocols are shown in Appendix C-A.

Configuration history. We define configuration history as a set of sequentially ordered membership requests according to the configuration number. The configuration history is a subset of the entire execution history. A configuration history only includes batches of requests where each batch consists of at least one membership request. The corresponding c number in each message is sequentially ordered. We additional require every replica to maintain the corresponding proof of delivery for each membership request, i.e., a message signed by Q_c replicas in M_c. A single configuration history can be verified by any correct replica to prove the existence of a configuration.

```
Initialization
  $c, M_c, chist$ {current configuration, membership, configuration history}
as a client/new replica
  func $ObtainConfig()$
    broadcast $\langle \textsc{Discover}, c\rangle$ to $\Pi$
    start a timer $\Delta$
    upon $\langle \textsc{Conf}, c', M_c', chist'\rangle$
      if $chist'$ is valid and $c' > c$
        $chist \leftarrow chist'$, $c \leftarrow c'$, $M_c \leftarrow M_c'$
    upon timeout($\Delta$)
      return $c, M_c$
as a replica
  upon $\langle \textsc{Discover}, c'\rangle$
    reply with $\langle \textsc{Conf}, c, M_c, chist\rangle$
```

Fig. 7: Configuration discovery: self-discovery.

Self-discovery. By default, we use a self-discovery approach shown in Fig. 7. To obtain the configuration of the system, a replica (or a client) p_i first sends a $\langle \textsc{Discover}, c\rangle$ request to the universe, where c is the latest configuration p_i is aware of. A timer is also started. Upon receiving a $\langle \textsc{Discover}, c'\rangle$ request, a replica replies with a $\langle \textsc{Conf}, c, M_c, chist\rangle$, where c is its current configuration, M_c is the members of configuration c, $chist$ is the configuration history. Upon receiving a Conf message with a valid $chist$, p_i updates its local configuration number, M_c, and $chist$. Upon time out of Δ, configuration discovery completes and the current c and M_c are returned.

Theorem VI.1. *Under the standard quorum assumption, Dyno achieves agreement V, total order, liveness, and consistent delivery.*

The correctness of Dyno is shown in Appendix C-B.

VII. Dyno with Stronger Agreement Properties

We study Dyno variants with stronger properties, i.e, V_1, V_2, V', V_1', and V_2'. In this section, we present two approaches for Dyno to achieve the V_1 agreement property. We first show that Dyno itself can achieve V_1 by making a stronger assumption. We then show Dyno-A, a construction based on Dyno, which achieves V_1 under the standard quorum assumption. Then we present Dyno-AC, which adds a simple constraint on top of Dyno-A to achieve V_2. We further show that the constructions of the protocol remain the same for Dyno with V, V_1, and V_2 to achieve V', V_1', and V_2', separately. The proofs of all protocols discussed in this section are shown in Appendix C-C.

Dyno with V_1 under G-correct assumption. We do not change the construction of the protocol to achieve the correctness but assume the G-correct assumption in Sec. III. Since we do not modify the protocol, the total order and liveness follow that of Dyno. The agreement property, however, can be greatly simplified.

Theorem VII.1. *Under the G-correct assumption, Dyno achieves agreement V_1, total order, liveness, and consistent delivery.*

Dyno-A with V_1. The assumption that there always exist at least $f + 1$ g-correct replicas is not practical as it requires one to know $max(f_0, f_1 \cdots)$, i.e., the maximum number of faulty replicas in the entire life of the system. Instead, we show another construction that achieves the V_1 agreement property without having to change the assumption of the system. In particular, we provide an addendum protocol on top of Dyno to achieve V_1.

```
upon timeout
  $c', M_{c'} \leftarrow ObtainConfig()$          {obtain a new configuration if any}
  if $c' = c$, $view\text{-}change()$                         {defined inFig. 6}
  else
    broadcast $\langle \text{UPDATE}, s, c, i \rangle$ to $M_c$
    upon $f_{c'} + 1$ matching $\langle \text{RESULT}, hist \rangle$
      deliver requests in $hist$
upon $\langle \text{UPDATE}, s, c', j \rangle$
  if $p_j \in M_{c'}$ and $c > c'$
    send $\langle \text{RESULT}, hist \rangle$ to $p_j$       {execution history greater than $s$}
```

Fig. 8: Dyno-A.

As shown in Fig. 8, we introduce additional procedures for each replica that falls behind to catch up with replicas in newer configuration. In particular, during the normal-case operation, each replica p_i still sets up a timer for the first request in its queue. If the request is not delivered before the timer expires, instead of directly triggering view changes, the replica first runs the configuration discovery protocol, obtains a new configuration number c', and the list of replicas $M_{c'}$. If $c' = c$, the replica starts the view change according to the procedures in Dyno. Otherwise, if c' is greater than c, the replica broadcasts an $\langle \text{UPDATE}, s, c, i \rangle$ message to all replicas in configuration c', where s is the sequence number of the last delivered request and c is its current configuration. When a replica p_i in c' receives an $\langle \text{UPDATE}, s, c', j \rangle$ message, it first verifies whether p_j is a valid replica in configuration c'. If the local execution history of p_i is longer (i.e., the sequence number of its last committed request is greater than s), p_i sends the execution history to p_j. Upon receiving $f_{c'} + 1$ matching $hist$, replica delivers the requests in $hist$. If a $\langle \text{REMOVE}, i \rangle$ request has been delivered, p_i directly leaves the system. Otherwise, p_i continues to participate in the protocol.

Theorem VII.2. *Under the standard quorum assumption, Dyno-A achieves agreement V_1, total order, liveness, and consistent delivery.*

The motivation of the addendum protocol is for a c-correct replica to obtain the execution history, even if the replica falls behind. In the case where a replica times out, it executes the configuration discovery protocol before it starts view change. If the replica falls behind, it obtains the execution history from other replicas. This ensures that any c-correct replica obtains the execution history and delivers the requests before it leaves the system (if applicable).

Dyno with V_2. Based on Dyno with G-correct assumption or Dyno-A, we could further add more constraints to replicas so that the protocol achieves V_2, creating Dyno-C and Dyno-AC, separately. In particular, V_2 further requires that any correct replica in configuration c also delivers a message if a correct replica in c delivers the messages.

The additional constraint is quite simple: any correct replica p_i leaves the system (if a leave request has been submitted) only after it delivers a $\langle \text{REMOVE}, i \rangle$ request. [2]

The constraint ensures that any correct replica in c delivers all the messages, even if it is removed immediately after c. Therefore, we require a replica to discover newer configuration(s) and obtain the execution history if it falls behind. This ensures that each correct replica delivers all the requests it *should* deliver according to the requirements of V_2.

Theorem VII.3. *Under the G-correct assumption, Dyno-C achieves agreement V_2, total order, liveness, and consistent delivery. Under the standard quorum assumption, Dyno-AC achieves agreement V_2, total order, liveness, and consistent delivery.*

Dyno with V', V_1', V_2'. There is no need to change the specification of the protocols to achieve V', V_1', and V_2'. Namely, Dyno achieves V', Dyno-A (or Dyno under the G-correct assumption) achieves V_1', and Dyno-C and Dyno-AC achieve V_2'.

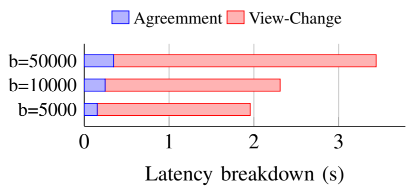
VIII. Implementation and Evaluation

Overview. We implement Dyno variants in Go using around 10,000 LOC. We implement an optimization for Dyno where a replica that joins the system starts to participate in the normal-case operation after the membership request is delivered. The state transfer is executed in parallel. All the requests after the delivery of the membership request, however, are executed after the state transfer is completed.

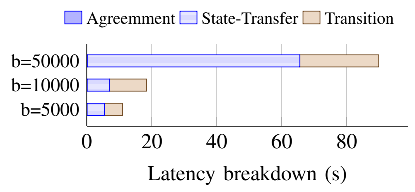
Our results show that Dyno-C and Dyno-AC achieve similar performance with Dyno, mainly because the additional constraints and addendum protocol mostly affect the behavior of replicas joining or leaving the system. Therefore, in this section, we focus on the performance of Dyno and Dyno-S. We compare Dyno with BFT-SMaRt [50], an open-source implementation of a variant of PBFT protocol written in Java. BFT-SMaRt supports reconfiguration where membership requests are issued by a separate view manager.

We deploy the protocols in a cluster using up to 30 servers. Each server has 16-core 2.3GHz CPU. We use f to represent the network size, where we use $3f + 1$ replicas in each experiment. Unlike previous protocols that mostly focus on benchmarks with small transactions, in our experiments, we set all transactions and reply messages to 100 bytes, as transactions in BFT applications (e.g., blockchains) are usually at least 100 bytes. Besides the number of replicas, we also vary the frequencies for garbage collection (i.e., checkpoint), denoted as *cp*, and the number of clients that submit transactions concurrently to the system. By default, replicas execute the checkpoint protocol upon delivering every 100 batches of requests.

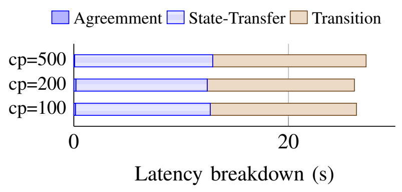
[2]It is worth mentioning that the pseudocode for Dyno already *enforces* such a constraint. In fact, such a constraint is not necessary in Dyno. According to the proof, a replica can leave after it is certain that the $\langle \text{REMOVE}, i \rangle$ request will be delivered, e.g., after it receives a prepare certificate.



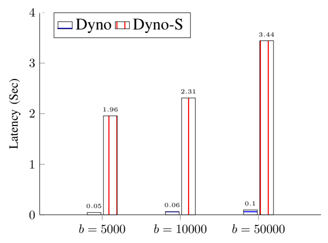
(a) Latency breakdown of a join request (excluding state transfer) for Dyno-S.



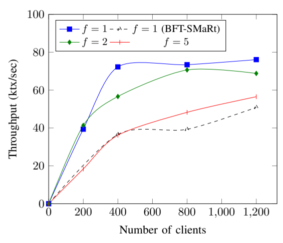
(b) Latency breakdown of a join request for Dyno (latency of agreement in all cases are lower than $100ms$ and not visible in the figure).



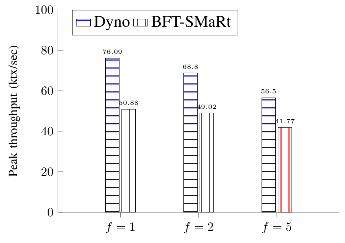
(c) Latency of a join request for Dyno and Dyno-S (latency of agreement in all cases are lower than $100ms$ and not visible in the figure).



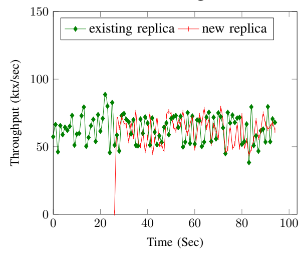
(d) Latency of membership requests (agreement only) for Dyno under different checkpoint frequency.



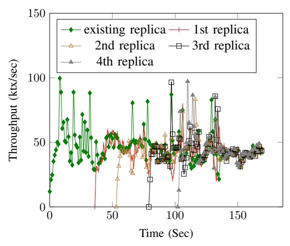
(e) Throughput of Dyno and BFT-SMaRt.



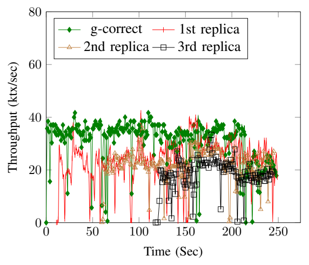
(f) Peak throughput of Dyno and BFT-SMaRt.



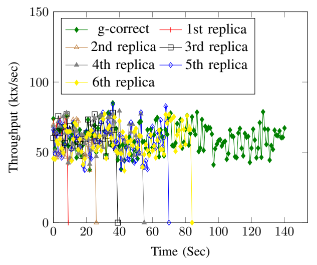
(g) Throughput of Dyno under a join request.



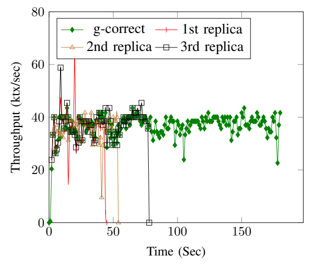
(h) Throughput of Dyno under multiple join requests.



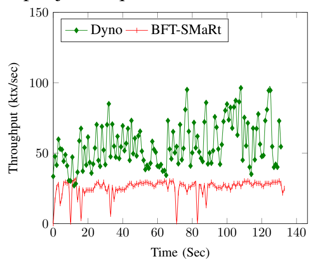
(i) Throughput of BFT-SMaRt under multiple join requests.



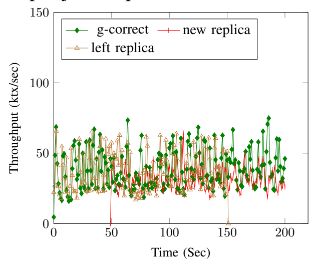
(j) Throughput of Dyno under multiple leave requests.



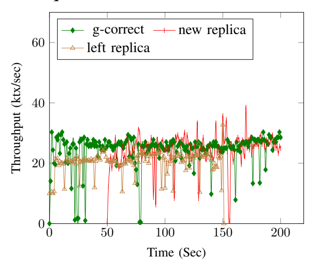
(k) Throughput of BFT-SMaRt under multiple leave requests.



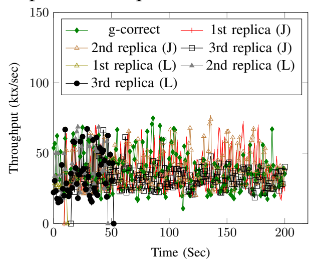
(l) Throughput of Dyno and BFT-SMaRt under multiple join and leave requests. System begins with 4 replicas.



(m) Throughput of Dyno under multiple join and leave requests. System begins with 10 replicas.



(n) Throughput of BFT-SMaRt under multiple join and leave requests. System begins with 10 replicas.



(o) Throughput of Dyno under frequent join and leave requests. J is for join and L is for leave.

Fig. 9: Performance of Dyno-S, Dyno, and BFT-SMaRt.

Latency. We evaluate the latency of membership requests for Dyno and Dyno-S. We set $f = 1$ and let one client submit regular requests continuously. We let a replica submit a membership request (join or leave) after 5000, 10000, and 50000 requests are submitted, separately (denoted as b in the figures). We show the latency for agreement only in Fig. 9d. Dyno has $50ms$ to $100ms$ latency on average. The results are similar to those of regular requests. In contrast, since Dyno-S runs a view change protocol upon every membership change, the latency is significantly higher. The latency breakdown for Dyno-S is shown in Fig. 9a. It can be observed that view change is the bottleneck, and the latency is higher if more requests are processed before the view change.

Since each new replica performs a state transfer after joining the system, we also assess the latency of state transfer for each of the experiments. We show the results for Dyno in Fig. 9b. We separate the latency for state transfer (network communication) and the latency for *transition* (the replica processes the historic client requests). As shown in the figure, the latency becomes significantly higher as b increases. This is expected since a large number of requests need to be synchronized during the state transfer. In contrast, the latency for agreement is almost negligible.

For the experiment with $b = 10000$, we also vary the checkpoint frequency and assess the latency of a join request. As shown in Fig. 9c, checkpoint frequencies do not have a direct impact on the latency. This is expected since each replica synchronizes all the historic transactions so the network bandwidth consumption is dominated by the state transfer.

Throughput and scalability. We evaluate the throughput of Dyno for $f = 1$, $f = 2$, and $f = 5$. As shown in Fig. 9e, when $f = 1$, the peak throughput of Dyno is 76 ktx/s, which is among the highest for partially synchronous BFT known so far. When f increases, the performance downgrades, as observed for almost all BFT protocols. We report the peak throughput of Dyno and BFT-SMaRt in Fig. 9f. Dyno achieves higher peak throughput than BFT-SMaRt, partly due to the efficiency of the underlying implementation.

Performance under membership requests. We assess the performance for membership requests for Dyno. We let $f = 1$ and let 400 clients submit regular requests concurrently to the system. We evaluate three different scenarios: 1) performance under join requests; 2) performance under leave requests; 3) performance under both join and leave requests.

Performance under join requests. We let one replica submit a join request and assess the throughput of both a g-correct replica (i.e., a replica that is correct since time 0) and the new replica. As shown in Fig. 9g, the system does not suffer from any performance degradation upon a join request.

For performance under multiple join requests, we begin with 4 replicas ($f = 1$). We then issue membership requests and add replicas one after another on a regular basis, until the system has 8 replicas. We show the throughput for a g-correct replica and every new replica that joins the system. As shown in Fig. 9h, the system suffers from performance degradation upon receiving every join request and resumes back to *normal* after a period of time. For instance, when the first replica joins, the throughput downgrades from 50 ktx/s on average to around 30 ktx/s, a 40% degradation. This is expected since the new replica performs state transfer while processing regular requests. After the state transfer completes, however, the throughput goes back to 50-60 ktx/s. As every replica joins, the overall system throughput degrades gradually. This is also expected, since the system has more replicas. When the system has 8 replicas, the overall throughput decreases from 50-60 ktx/s to 40-50 ktx/s.

We match most of the system configuration parameters and evaluate the performance of BFT-SMaRt in the same setting. The results are shown in Fig. 9i. Unlike Dyno, in BFT-SMaRt, the throughput of replicas that join the system is consistently lower than that of existing ones. Furthermore, the overall throughput degradation as replicas join for Dyno is consistently lower than that of BFT-SMaRt.

Performance under leave requests. We begin with 10 replicas ($f = 3$) and then let 6 replicas leave one after one another. The throughput of the system is shown in Fig. 9j. In contrast to the prior case, the throughput is more stable, mostly because a replica can directly leave the system upon the delivery of the remove request. The performance for BFT-SMaRt is similar, as shown in Fig. 9k, besides that the overall throughput of BFT-SMaRt is about 30%-40% lower than that of Dyno.

Performance under multiple join and leave requests. We also assess the performance under both join and leave requests. In particular, we trigger 3-5 random join or leave requests under 30-sec intervals. We first begin with 4 replicas and evaluate the performance for both Dyno and BFT-SMaRt. We present the performance of a g-correct replica in Fig. 9l. The performance is similar to that under only join or leave requests. BFT-SMaRt may hit low (close to 0) throughput during some requests. In contrast, the performance of Dyno is in general more stable.

We let the system have $f = 3$ (10 replicas) in the beginning and conduct the same experiment again. We present the results for Dyno in Fig. 9m and BFT-SMaRt in Fig. 9n. We present the performance of a g-correct replica, a replica that joins the system, and a replica that leaves the system. The results for both protocols, despite the fact that the throughput is in general lower, are similar to that when $f = 1$.

We also evaluate Dyno under frequent membership requests, where we begin with 10 replicas and let replicas frequently join and leave under random intervals. As shown in Fig. 9o, with frequent membership requests, the throughput of the system tends to be more turbulent, as replicas have to perform frequent state transfer for newly joined replicas.

IX. Conclusion

We study dynamic BFT protocols, where replicas may join and leave the system. We formally define the security definitions for dynamic BFT and present different but meaningful variants. We present Dyno, a highly efficient dynamic BFT protocol. We show, with a up to 30-server deployment, that Dyno is efficient, handling membership requests with low cost.

Acknowledgment

We thank Xiaoyun Wang, Xiao Sui, Baohan Huang, and the anonymous reviewers for the help and comments for the paper. Sisi is also with Shandong Institute of Blockchain. Sisi was supported in part by Tsinghua Independent Research Program, Shandong Key Research and Development Program under grant No. 2020ZLYS09 and National Key Research and Development Program of China under grant No. 2018YFA0704701. Haibin was supported in part by Teli Young Scholar program.

References

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *S&P*, 2020.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the Acm*, 58(2):96–99, 2009.

[3] E. Alchieri, A. Bessani, F. Greve, and J. Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, 2017.

[4] Y. Amir, C. Nita-Rotaru, S. Stanton, and G. Tsudik. Secure spread: An integrated architecture for secure group communication. *TDSC*, 2(3):248–261, 2005.

[5] Y. Amir and J. Stanton. The spread wide area group communication system. Technical report, Citeseer, 1998.

[6] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *ATC*, 2013.

[7] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *TOCS*, 1987.

[8] G. Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *PODC*, pages 154–162. ACM, 1984.

[9] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.
[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.
[11] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *CSUR*, 33(4):427–469, 2001.
[12] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: location-aware membership management for large-scale distributed systems. In *ATC*, 2009.
[13] A. Das, I. Gupta, and A. Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *DSN*, 2002.
[14] S. Duan, K. Levitt, H. Meling, S. Peisert, and H. Zhang. ByzID: Byzantine fault tolerance from intrusion detection. In *SRDS*, 2014.
[15] S. Duan, H. Meling, S. Peisert, and H. Zhang. BChain: Byzantine replication with high throughput and embedded reconfiguration. In *OPODIS*, pages 91–106, 2014.
[16] S. Duan, M. K. Reiter, and H. Zhang. Beat: Asynchronous bft made practical. In *CCS*, pages 2028–2041. ACM, 2018.
[17] S. Duan and H. Zhang. Pace: Fully parallelizable bft from reproposable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2022.
[18] E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, 2015.
[19] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
[20] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y. A. Pignolet, D. Seredinschi, and A. Tonkikh. Dynamic Byzantine reliable broadcast. In Q. Bramas, R. Oshman, and P. Romano, editors, *OPODIS*, 2020.
[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, 2010.
[22] L. Jehl and H. Meling. Asynchronous reconfiguration for paxos state machines. ICDCN, 2014.
[23] L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, 2015.
[24] H. D. Johansen, R. Van Renesse, Y. Vigfusson, and D. Johansen. Fireflies: A secure and scalable membership and gossip service. *TOCS*, 33(2):1–32, 2015.
[25] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-smith. The SecureRing protocols for securing group communication. In *HICSS*, 1998.
[26] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security*, pages 279–296, 2016.
[27] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger. *S&P*, 2018.
[28] P. Kuznetsov and A. Tonkikh. Asynchronous reconfiguration with byzantine failures. DISC, 2020.
[29] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *TOPLAS*, 6(2):254–280, 1984.
[30] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.
[31] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC*, 2009.
[32] J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *DSN*, 2007.
[33] C. Liu, S. Duan, and H. Zhang. Epic: Efficient asynchronous bft with adaptive security. In *DSN*, 2020.
[34] 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 Y. Berbers and W. Zwaenepoel, editors, *EuroSys*, pages 103–115, 2006.
[35] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxbood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.
[36] J. P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *DSN*, 2004.
[37] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *ICDCS*, pages 56–65. IEEE, 1994.
[38] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.
[39] M. K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *CCS*, pages 68–80, 1994.
[40] M. K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.
[41] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 2003.
[42] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Middleware*, 2009.
[43] R. V. Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. *OSDI*, 2004.
[44] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. 2003.
[45] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. Schultz. Automatic reconfiguration for large-scale reliable storage systems. *TDSC*, 2012.
[46] A. Schiper. Dynamic group communication. *Distributed Comput.*, 18(5):359–374, 2006.
[47] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. 1993.
[48] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR*, 22(4):299–319, 1990.
[49] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *ATC*, 2011.
[50] J. Sousa, E. Alchieri, and A. Bessani. State machine replication for the masses with bft-smart. In *DSN*, pages 355–362, 2014.
[51] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Veríssimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distributed Syst.*, 21(4):452–465, 2010.
[52] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala. Stable and consistent membership at scale with rapid. In *ATC*, 2018.
[53] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*, 2009.
[54] X. Wang, S. Duan, J. Clavin, and H. Zhang. BFT in blockchains: From protocols to use cases. *ACM Computing Surveys (CSUR)*, 2021.
[55] M. Yin, D. Malkhi, M. Reiterand, G. G. Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.
[56] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: A fast blockchain protocol via full sharding. In *CCS*, pages 931–948, 2018.

Appendix A
Normal-Case Operation Oracle of Dyno

We describe the normal-case operation of Dyno and how the *init*() and *deliver*() events are triggered. In particular, we use Bracha's broadcast paradigm and use PBFT notations to present the protocol. The pseudocode is illustrated in Fig. 10.

```
Initialization
  $v, c, M, TM$ {view, configuration, membership, temporary membership}
▷ events
  upon receiving a valid $m = \langle \textsc{Pre-prepare}, v', c', s, batch \rangle$
    if $\langle \textsc{Add}, j, m \rangle \in batch$
      $TM \leftarrow TM \cup \{p_j\}$
    if non-primary, $init(batch)$
    broadcast $\langle \textsc{Prepare}, v, c, s, h(batch) \rangle$ to $TM$
  upon receiving $2f_c + 1$ matching $\langle \textsc{Prepare}, v, c, s, \delta \rangle$
    $prepared(\delta, v, c, s) \leftarrow true$
    broadcast $\langle \textsc{Commit}, v, c, s, \delta \rangle$ to $TM$
  upon receiving $f_c + 1$ matching $\langle \textsc{Commit}, v, c, s, \delta \rangle$
    broadcast $\langle \textsc{Commit}, v, c, s, \delta \rangle$ to $TM$
  upon receiving $2f_c + 1$ matching $\langle \textsc{Commit}, v, c, s, \delta \rangle$
    $deliver(batch)$ where $h(batch) = \delta$
▷ oracle functions
  func $init(batch)$                                          {$init()$ event}
    if leader, broadcast $\langle \textsc{Pre-prepare}, v, c, s, batch \rangle$ to $TM$
    {switch to procedures shown in Fig. 4}
  func $deliver(batch)$                                       {$deliver()$ event}
    {switch to procedures shown in Fig. 4}
```

Fig. 10: Normal-case operation and how the *init*() and *deliver*() events are triggered.

The pseudocode specifies the workflow for replicas in the current configuration c and *pending* replicas, i.e., replicas that request to join the system. Replicas run a Bracha's broadcast protocol and replicas that request to join the system act as learners before the corresponding join requests are delivered.

In particular, for replicas in M_c, the leader first triggers the *init*() event by broadcasting a $\langle$PRE-PREPARE$, v, c, s, batch\rangle$ message to TM, where TM is the temporary members. (TM is updated according to Fig. 5). Upon receiving a $\langle$PRE-PREPARE$, v', c', s, batch\rangle$ message, a replica p_i verifies the following: 1) $v' = v$; 2) the signatures of client requests in $batch$ are valid; 3) p_i has not accepted another $\langle$PRE-PREPARE$\rangle$ message with sequence number s; 4) s is within a valid range. A non-primary replica also triggers the $init(batch)$ event accordingly. After that, p_i broadcasts a $\langle$PREPARE$, v, c, s, h(batch)\rangle$ to all replicas in TM where $h(batch)$ is the hash of the batch. Replica p_i waits until it receives $2f_c + 1$ valid $\langle$PREPARE$\rangle$ messages, i.e., p_i has previously accepted a $\langle$PRE-PREPARE$, v, c, s, batch\rangle$ message. We say that *batch* is *prepared* by the replica. After that, p_i broadcasts a $\langle$COMMIT$, v, c, s, \delta\rangle$ to all replicas in TM. If a replica receives $f_c + 1$ $\langle$COMMIT$\rangle$ messages but has not previously broadcast any $\langle$COMMIT$\rangle$ message, it also broadcasts a $\langle$COMMIT$, v, c, s, \delta\rangle$ messages. A set of either $2f_c + 1$ $\langle$PREPARE$\rangle$ messages or $f_c + 1$ $\langle$COMMIT$\rangle$ messages serve as a prepare certificate. Note that this is different from protocols such as PBFT but crucial in our protocol. The certificate also serves as a *proof of delivery*. Finally, if a replica receives $2f_c + 1$ $\langle$COMMIT$, v, c, s, \delta\rangle$ messages, it triggers the *deliver*($batch$) event.

No that during the view changes, a message in the form of $\langle$VIEW-CHANGE$, v, c, \mathcal{C}, \mathcal{P}, \mathcal{PP}, i\rangle$ carries $\mathcal{P}$ and $\mathcal{PP}$ which are related to the normal-case operation oracle. The $\mathcal{P}$ is a set of prepare certificates for requests with sequence number greater than C, the $\mathcal{PP}$ is a set of $\langle$PRE-PREPARE$\rangle$ messages where each message includes at least one membership request.

APPENDIX B
CONFIGURATION DISCOVERY OPTIONS

We present two alternatives options for configuration discovery: lazy discovery and configuration master.

Lazy discovery. Lazy discovery delays the discovery of the configuration after the delivery of the request. In particular, to obtain the latest configuration, a new replica or a client p_i directly obtains the universe, i.e., Π. In this case, every time a client or a replica receives a messages, it must verify the configuration history accordingly.

Configuration master. Both self-discovery and lazy discovery require all clients and new replicas to broadcast to all replicas in the universe. Alternatively, we could build a standalone configuration service all replicas/clients can query to obtain the latest configuration of the system, as shown in Fig. 11. We assume that all replicas and clients know the identities of all replicas in the configuration master. The configuration master can be built as one or a subset of all replicas in the system. We let the master passively learn the latest configuration. In particular, if the configuration changes, replicas send a set of $2f_c + 1$ $\langle$COMMIT$\rangle$ messages together with the corresponding $\langle$PRE-PREPARE$\rangle$ message to the configuration master. The $2f_c+1$ $\langle$COMMIT$\rangle$ messages serve a a proof of delivery and the $\langle$PRE-PREPARE$\rangle$ messages can be used to verify the membership request(s). In this way, the configuration master also obtains the entire configuration history.

Note that the configuration master does not have to be replicated using SMR. This is mainly because the entire configuration history is totally ordered. Therefore, any configuration history can be self validated.

```
**Initialization**
  $c, M_c, chist$      {configuration, membership, and configuration history}
**as a client/new replica**
  **func** $ObtainConfig()$
    send $\langle$QUERY$, i\rangle$ to CMaster
    **upon** $\langle$CONF$, c', M'_c, chist'\rangle$
      **if** $chist'$ is valid and $c' > c$
        $chist \leftarrow chist'$, $c \leftarrow c'$, $M_c \leftarrow M'_c$
        **return** $c, M_c$
**as a configuration master**
  **upon** $\langle$QUERY$, j\rangle$
    send $\langle$CONF$, c, M_c, chist\rangle$ to $p_j$
  **upon** $M = 2f_c + 1$ $\langle$COMMIT$, v, c, s, h(batch)\rangle$ messages
    $chist \leftarrow chist \cup M$, update $c$ and $M_c$
```

Fig. 11: Configuration discovery: configuration master.

APPENDIX C
PROOFS

A. Proof of Configuration Discovery

Lemma C.1. *A configuration history chist can be verified by any replica/client.*

Proof: Let *chist* be a configuration history from configuration 0 to c. All the replicas know the members in M_0 according to our assumption. The *chist* consists of all the proofs of delivery monotonically ordered by configuration numbers. Let the sets of proofs of delivery be $\{cert_i\}$ s.t. $0 \leq i \leq c$ The proof of delivery numbered by configuration 0 consists of signatures from Q_0 replicas in M_0. Therefore, all replicas can verify $cert_0$. Furthermore, every replica/client can obtain M_1 from $cert_0$ since it consists of the information of new replicas and replicas that are removed in the end of configuration 0. Similarly, $cert_1$ consists of Q_1 signatures, so every replica/client can verify it accordingly. It is then straightforward to see that *chist* can be verified by any replica. □

Lemma C.2. *The configuration history is totally ordered.*

Proof: We assume that there are two valid configuration histories $chist^1$ and $chist^2$, both from configuration 0 to c. Let c' be the first configuration where the proofs of delivery are inconsistent, i.e., 1) $0 < c' \leq c$; 2) M_j is the same in both $chist^1$ and $chist^2$ for $0 \leq j < c'$. Let $cert^1_{c'} \in chist^1$ and $cert^2_{c'} \in chist^2$. $cert^1_{c'}$ consists of $Q_{c'-1}$ signatures from $M_{c'-1}$. Similarly, $cert^2_{c'}$ consists of $Q_{c'-1}$ signatures from $M_{c'-1}$. The two quorums of replicas have at least one correct replica in common. Therefore, at least one correct replica has signed both the requests such that $chist^1$ and $chist^2$ have inconsistent proofs of delivery, a contradiction. □

Lemma C.3. *Self-discovery achieves configuration discovery.*

Proof. Let c be the latest configuration. A replica p_i broadcasts a $\langle$DISCOVER$, c'\rangle$ to all replicas in the universe. In other words, all replicas in M_c will eventually receive $\langle \text{DISCOVER}, c' \rangle$. All correct replicas in M_c will reply with $\langle \text{CONF}, c, M_c, chist \rangle$ where $chist$ is a valid configuration history. It is straightforward to see that p_i eventually receives a valid configuration. If $ObtainConfig()$ is queried for a sufficiently large number of times, the latest configuration and members in the configurations will be returned. □

Lemma C.4. *Lazy discovery achieves configuration discovery.*

Proof. Π includes replicas in the latest configuration. □

Lemma C.5. *Configuration master achieves configuration discovery.*

Proof. Following Lemma C.3, any correct replica in the configuration master is able to obtain members from a valid configuration since it follows the procedure of self-discovery. Therefore, any client/replica queries the configuration master is able to obtain members from a valid configuration if the configuration master is correct. □

Theorem C.1. ***(Configuration discovery)*** *If $ObtainConfig()$ is queried for a sufficiently large number of times, members from the latest configuration c will be returned, if there exists any such c.*

Proof. Correctness follows from Lemma C.3-C.5. □

B. Proof of Dyno

We prove correctness of the protocol based on the default definitions of correctness.

Lemma C.6. *During the view change, if the leader in view v is correct and timers are properly set up, correct replicas will eventually enter view v.*

Proof. Let the latest configuration of the system be c, i.e., at least one c-correct replica p_i has installed c by delivering some membership request m. The replica previously received Q_{c-1} matching $\langle \text{COMMIT} \rangle$ messages for m. Among the replicas, at least $Q_{c-1} - f_{c-1} \geq f_{c-1} + 1$ are correct (set S). There are two cases from configuration $c - 1$ to c: at least one replica joins the system; no replica has joined the system. If no replica has joined the system, all replicas are aware of the replicas in c. Therefore, the new leader eventually receives $f_c + 1$ $\langle \text{VIEW-CHANGE} \rangle$ messages. We now only need to show the first case.

In particular, any new replica that joins the system completes state transfer with $2f_{c-1} + 1$ replicas in the system. Among the $2f_{c-1} + 1$ replicas, at least one correct replica in S is included according to the quorum intersection. Therefore, the new replica must have maintained consistent execution history as replicas in S. Furthermore, there are two sub-cases considering all the replicas that join the system in c: none of the new replicas is correct; at least one of the new replica is correct. In the first case, the new leader will receive $f_c + 1$ $\langle \text{VIEW-CHANGE} \rangle$ messages, as all correct replicas in $c - 1$ are aware of M_c. In the second sub-case, at least one new replica p_i is correct. It will broadcast a $\langle \text{VIEW-CHANGE} \rangle$ message to all replicas in c. If a replica has not previously installed c but receives the message from p_i, the replica will also send its $\langle \text{VIEW-CHANGE} \rangle$ message to M_c. Thus, all correct replicas will receive $f_c + 1$ matching $\langle \text{VIEW-CHANGE} \rangle$ messages. Eventually all correct replicas in c will receive $\langle \text{VIEW-CHANGE} \rangle$ messages, including the new leader in view v. □

Lemma C.7. *If a correct replica p_i delivers a request m with sequence number s in view v and configuration c, and another correct replica delivers m' with sequence number s in view v' and c where $v' > v$, $m = m'$.*

Proof. If p_i delivers m in view v and c, at least Q_c replicas have previously broadcast $\langle \text{COMMIT} \rangle$ message. Among the replicas, $Q_c - f_c$ replicas are correct. The correct replicas all have previously stored prepare certificates (either a set of $2f_c + 1$ $\langle \text{PREPARE} \rangle$ messages or $f_c + 1$ $\langle \text{COMMIT} \rangle$ messages) for m with s. If view change occurs, according to Lemma C.6, replicas will eventually move to a view v' where the leader is correct. There are two cases for configuration c: c is the latest configuration; the latest configuration is c' where $c' > c$.
Case (1): The new leader in view v' receives Q_c matching $\langle \text{VIEW-CHANGE} \rangle$ messages. In the new view, the leader assigns each sequence number s with a m if there is a prepare certificate for s. If a replica accepts s with a different m' in the new view, at least one $\langle \text{VIEW-CHANGE} \rangle$ messages includes a valid prepare certificate for m'. The certificate consists of at least $Q_{c'}$ $\langle \text{PREPARE} \rangle$ messages or $f_{c'} + 1$ $\langle \text{COMMIT} \rangle$ messages for $c' \leq c$. According to the protocol, a correct replica will not send a $\langle \text{PREPARE} \rangle$ message for request m' if it has already sent a $\langle \text{PREPARE} \rangle$ message for some request m. Therefore, at least one correct replica has sent a $\langle \text{PREPARE} \rangle$ message for request m' in configuration c' and $\langle \text{PREPARE} \rangle$ message for request m in configuration c', both in the same view, a contradiction.
Case (2): Since at least one c'-correct replica p_j installs c', it has delivered at least one membership request, i.e., $\langle \text{ADD} \rangle$ and/or $\langle \text{REMOVE} \rangle$ requests. Therefore, at least $Q_{c'-1}$ replicas have previously broadcast $\langle \text{COMMIT} \rangle$ messages for the membership request(s), among which $Q_{c'-1} - f_{c'-1}$ are $(c' - 1)$-correct. There are two cases: $c = c' - 1$, $c < c' - 1$.

If $c = c' - 1$, $Q_c = Q_{c'-1}$, at least one c-correct replica in the intersection of Q_c and $Q_{c'-1}$ has delivered m before delivering the membership request(s). Similar to the proof shown in Case (1), $m = m'$.

We now show the case for $c < c' - 1$. Assume that the correct replica p_j delivers m' in view v' and configuration c', p_j receives $Q_{c'} - f_{c'}$ matching messages, among which at least $f_{c'} + 1$ are correct. Let the set of replicas be S. We conclude that none of the replicas in S is g_c-correct. This is because a g_c-correct replica that enters configuration c' must have delivered at least one batch of requests (including a membership request) after it delivers m. Now we consider a correct replica p_k in S. We assume p_k joins the system in configuration c'' where $c < c'' < c'$. Replica p_k must have delivered a request m'' that includes $\langle \text{ADD}, k \rangle$. Before it continues to process requests in configuration c'', it completes a state transfer with $Q_{c''}$ replicas in configuration c'', among which $f_{c''} + 1$ replicas are correct. The correct replicas must have sent an execution history with valid prepare certificate for m' with sequence number s. A correct replica will not send $\langle$PREPARE$\rangle$ message for m' if it has sent $\langle$PREPARE$\rangle$ message for m or has received a prepare certificate. Therefore, at least one correct replica has sent $\langle$PREPARE$\rangle$ message for both m and m', or at least one correct replica has received a valid prepare certificate for m and later sent $\langle$PREPARE$\rangle$ message for m', a contradiction. □

Theorem C.2. *(Agreement V) If a **correct** replica in c delivers a request m, then every g_c-correct replica eventually delivers m.*

Proof. We show that if a correct replica p_i delivers a regular request m, all g_c-correct replicas will deliver m, with the same sequence number s. We prove the theorem under two cases for configuration c: 1) c is the latest configuration; 2) c is not the latest configuration.
Case (1): If p_i delivers m, at least $2f_c + 1$ replicas have previously broadcast $\langle$COMMIT$\rangle$ messages, among which at least f_c+1 replicas are correct. If view change does not occur, all replicas eventually receive $f_c + 1$ $\langle$COMMIT$\rangle$ messages. If they have not previously sent one, they will also broadcast $\langle$COMMIT$\rangle$ messages. Therefore, all correct replicas will eventually collect Q_c matching $\langle$COMMIT$\rangle$ messages and deliver m. If view change occurs, Lemma C.7 shows that replicas will only deliver m with sequence s in the new view. Accordingly, any g_c-correct replica eventually delivers m.
Case (2): Let $c' < c$ be the latest configuration, There are two sub-cases: at least Q_c c-correct replicas are still in c'; fewer than Q_c replicas that are still c'-correct. The correctness of the first sub-case follows case (1). We now show the second sub-case. In this case, at least one c-correct replica has left the system in configuration c'' where $c < c'' < c'$. In other words, at least one correct replica has delivered $\langle$REMOVE$\rangle$ request and at least $Q_{c''}$ replicas have previously broadcast $\langle$COMMIT$\rangle$ messages, among which $Q_{c''} - f_{c''}$ are c''-correct. Any c''-correct replica maintains the entire execution history. This is because all g_c-correct replicas maintain the entire execution history. Furthermore, any new replica in the system starts to participate in the protocol after it completes the state transfer. Therefore, at least $Q_{c''} - f_{c''}$ c''-correct replicas include m in their execution history together with a valid prepare certificate. Similarly, if $c' > c''$, any c''-correct replica that has delivered the membership request must have included m in its execution history. If the number of g_c-correct replicas is greater than $Q_{c'} - f_{c'}$ and m has not been delivered for a sufficiently long time, view change will occur. As shown in Lemma C.7, in the new view, replicas will only accept m with sequence number s. Eventually, g_c-correct replicas will accept m. If the number of g_c-correct replicas is lower than $Q_{c'} - f_{c'}$ and replicas still continue to process requests, g_c-correct replicas that have not delivered m will catch up with other correct replicas after the next view change and eventually deliver m. □

Lemma C.8. *In the same view v, if a correct replica p_i delivers a request m with sequence number s in configuration c, and another correct replica p_j delivers a request m' with sequence number s in configuration c' where $c' \geq c$, $m = m'$.*

Proof. If p_i delivers m in c, it receives Q_c matching $\langle$COMMIT$\rangle$ messages. If p_j delivers m' in c', it receives $Q_{c'}$ matching $\langle$COMMIT$\rangle$ messages. If $c = c'$, the correctness simply follows from static BFT. We now show the correctness for $c' > c$. Without loss of generality, we consider $c' = c+1$. Correctness for $c' > c + 1$ can be proved by induction.

There are three cases from c to c': new replicas join the system from c to c'; some replicas leave the system from c to c'; multiple replicas join and leave from c to c'.
Case (1): We first consider the case where one replica joins in configuration c', i.e., $Q_{c'} = Q_c$ or $Q_{c'} = Q_c+1$. If $Q_c = Q_{c'}$, correctness simply follows. If $Q_{c'} = Q_c + 1$, $Q_c + 1$ replicas have sent $\langle$COMMIT$\rangle$ messages for m' in c'. Since Q_c replicas have sent $\langle$COMMIT$\rangle$ messages for m in c, the two quorums in total have size $2Q_c + 1$. We also know that $Q_c = \lceil \frac{n_c+f_c+1}{2} \rceil$, where n_c is the number of replicas in configuration c and f_c is the number of failures the system can tolerate. Since configuration c' has $n_c + 1$ replicas, the two quorums have at least $2Q_c + 1 - (n_c + 1) = 2\lceil \frac{n_c+f_c+1}{2} \rceil - n_c \geq f_c + 1$ replicas in total. In our system, any new replica that joins the system participates in the protocol after it completes state transfer from replicas in c. Therefore, if any of the $f_c + 1$ correct replicas is a new replica, it will not accept m' if m is included in its execution history. We also assume that $n_c \geq 3f_c + 1$. Therefore, at least one correct replica has sent $\langle$COMMIT$\rangle$ message for m in c and m' in c', a contradiction. For the case where multiple replicas join, n_c becomes $n_c + l$ and Q_c becomes $Q_c + q$ where $0 \leq q \leq l$ (concretely, q is bounded by $\lceil \frac{l}{3} \rceil$). The proof is similar to the case for $l = 1$.
Case (2): If l replicas are removed, n_c becomes $n_c - l$ and Q_c becomes $Q_c - q$ where $0 \leq q \leq l$. Consider the worst case where Q_c becomes $Q_c - l$, there are $2Q_c - l = 2Q_{c'} + l$ replicas in total for any two quorums. There are $2Q_{c'} + l - (n_{c'} + l) \geq f_{c'} + 1$ correct replicas in common. Thus, at least one correct replica has sent conflicting messages.
Case (3): We know that Q_c replicas have broadcast m and $Q_c - f_c$ c-correct replicas (set S) have maintained valid prepare certificates. If p_j delivers m' in c', at least $Q_{c'} - f_{c'}$ correct replicas have previously delivered m'. If there exists a correct replica p_i in $Q_{c'} - f_{c'}$ that is not in S, p_i must have joined the system in c as a correct replica will not deliver both m and m' with the same sequence number. Before p_i participates in the protocol, it completes state transfer with Q_c replicas in configuration c, i.e., all delivered requests before $\langle$ADD$, i\rangle$ request. If p_i delivers m', it does not have m in the execution history. Thus, none of Q_c replicas has sent a valid prepare certificate for m during state transfer, a contradiction with the fact that Q_c replicas have sent PREPARE messages for m. □

Lemma C.9. *A correct replica p_i delivers a request m with sequence number s, configuration c, and view v. Another correct replica p_j delivers a request m' with sequence number s in configuration c' and view v' where $c' \geq c$ and $v' > v$. Then $m = m'$.*

Proof. Without loss of generality, we let $c' = c + 1$ and let c' be the latest configuration, as the case for $c' > c + 1$ can be proved in the same way. Let the configuration changes in view v'' s.t. $v \leq v'' \leq v'$. When configuration changes in v'', at least one correct replica p_k in both M_c and $M_{c'}$ has delivered m. This is because from view v to the beginning of view v'', the configuration does not change. From configuration c to c', the view v'' does not change. Therefore, if p_i delivers m and p_k delivers m'', this is a contradiction with Lemma C.7 or Lemma C.8. Furthermore, from view v'' to v', configuration does not change, so if p_j delivers m' with s also in the configuration c', a contradiction with Lemma C.7. □

Theorem C.3. ***(Total Order)*** *If a correct replica in configuration c delivers a message m before delivering m', then another correct replica in configuration c delivers a message m' only after it has delivered m.*

Proof. The correctness within the same view is shown in Lemma C.8 and correctness across views is in Lemma C.9. □

Theorem C.4. ***(Same configuration delivery)*** *If a correct replica p_i (resp. p_j) delivers m in configuration c^i (resp. c^j), then $c^i = c^j$.*

Proof. For each correct replica p_i (resp. p_j), it delivers a request upon receiving $2f_{c^i} + 1$ (resp. $2f_{c^j} + 1$) matching $\langle \textsc{Commit}, v, c, s, h \rangle$ messages. It is straightforward to see that $c^i = c^j$ since the $\langle \textsc{Commit} \rangle$ messages have a matching configuration number c. □

Lemma C.10. *A client eventually obtains a valid configuration c from the configuration discovery protocol such that at least one g_c-correct replica in M_c has installed the latest configuration.*

Proof. A client repeats the $submit()$ function and obtains configuration until it obtains a valid reply. Thus, the client eventually obtains a valid configuration c (according to Theorem C.1) such that at least one g_c-correct replica is in M_c. □

Theorem C.5. ***(Liveness)*** *If a correct client submits a request m, then a correct replica in some configuration c eventually delivers m.*

Proof. We already show in Theorem C.2 that if p_i delivers m, all g_c-correct replicas will eventually deliver m. Consider c' is the latest configuration, there are two cases: there exist at least $f_c + 1$ g_c-correct replicas in c'; there are fewer than $f_c + 1$ g_c-correct replicas in c'. In the first case, the client will eventually receive $f_c + 1$ matching replies.

We now show the second case where there are fewer than $f_c + 1$ g_c-correct replicas in c'. According to Lemma C.1, a configuration history can be verified by the client. Therefore, if the client obtains a configuration c such that at least one g_c-correct replica in M_c has installed the latest configuration, the request will eventually be broadcast to all replicas. Eventually in some configuration, m is included in the queue of the leader and then processed. According to the agreement property, the request will eventually be delivered by all g_c-correct replicas. □

Theorem C.6. ***(Consistent delivery)*** *A correct client submitting m will deliver a correct response which is consistent with the state in com configuration where m is delivered.*

Proof. A correct client completes a request if it has received $f_c + 1$ matching replies. If the client previously submitted the request in c, it completes the request. Otherwise, the client verifies the configuration history. According to the total order and agreement properties, any correct replica will execute and deliver m with following the same order. Therefore, all correct replicas will generate matching response to the client. □

C. Proof of Dyno, Dyno-A, Dyno-AC

Theorem C.7. ***(Dyno achieves agreement V_1 under G-correct assumption)*** *If a correct replica in configuration c delivers a request m, then every* c*-correct replica eventually delivers m.*

Proof. According to Theorem C.2, if a correct replica delivers m, then every g_c-correct replica eventually delivers m. There are at least $f_c + 1$ g_c-correct replicas according to the assumption. Therefore, all c-correct replicas will eventually receive $f_c + 1$ $\langle \textsc{Commit} \rangle$ messages. Every correct replica broadcasts a $\langle \textsc{Commit} \rangle$ message if it receives $f_c + 1$ matching messages. Therefore, all c-correct replicas will eventually receive $2f_c + 1$ $\langle \textsc{Commit} \rangle$ messages and deliver m. □

Theorem C.8. ***(Dyno-A achieves agreement V_1)*** *If a correct replica in c delivers a request m, then every* c*-correct replica eventually delivers m.*

Proof. According to Theorem C.2, if a correct replica delivers m, then every g_c-correct replica eventually delivers m. Let the latest configuration of the system be c'. We show that a c-correct replica p_i that has not delivered m will eventually deliver m. If fewer than $2f_c+1$ replicas are still correct, replica p_i will eventually time out. p_i will query the configuration service and obtain c'' s.t. $c'' \leq c'$. The case for $c'' = c'$ is trivial. If $c'' < c'$ and there are more than $f_{c''} + 1$ $g_{c''}$-correct replicas, they will send correct $hist$ to p_i. We know that m is included in $hist$ of correct replicas in c''. Thus, p_i will deliver m. If there are fewer than $f_{c''} + 1$ $g_{c''}$-correct replicas, replica p_i will continue querying $ObtainConfig()$ and eventually obtain c'. According to the protocol, p_i will eventually obtain the request history and deliver m. □

Theorem C.9. ***(Dyno-AC achieves agreement V_2)*** *If a correct replica delivers a request m in c, then every correct replica in c eventually delivers m.*

Proof. According to Theorem C.8, every c-correct replica will deliver m. If a correct replica p_i leaves the system, it must have already delivered $\langle \textsc{Remove}, i \rangle$. Before p_i delivers $\langle \textsc{Remove}, i \rangle$, it must have delivered m since membership requests are delivered after regular requests. □