IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 June 2022, accepted 25 August 2022, date of publication 29 August 2022, date of current version 2 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3202627

RESEARCH ARTICLE

Dynamic Byzantine Broadcast in Asynchronous Message-Passing Systems

JING LI[1], TIANMING YU[1], YE WANG[2,3], AND ROGER WATTENHOFER[3]
[1]School of Statistics, Jiangxi University of Finance and Economics, Nanchang 330000, China
[2]Department of Computer and Information Science, University of Macau, Macau, China
[3]Department of Information Technology and Electrical Engineering, ETH Zürich, 8092 Zürich, Switzerland

Corresponding author: Ye Wang (wangye@um.edu.mo)

ABSTRACT The *reconfiguration* problem is considered a key challenge in distributed systems, especially in *dynamic* asynchronous message-passing systems. To keep the data reliability and availability in long-lived systems, any protocols should support reconfigurations, to dynamically add resources, or remove old and slow machines with newer faster ones. Previous results in reconfigurations either rely on consensus, or study the problem restricted to crash failures only. However, it is difficult to argue that real-world systems experience crash failures only. In this paper, we study the dynamic reconfiguration problem in fully asynchronous message-passing systems with Byzantine faults. We first specify dynamic Byzantine broadcast, and then specify a clean and explicit liveness condition. We show that dynamic Byzantine broadcast is solvable by presenting a dynamic Byzantine consistent broadcast algorithm and a dynamic Byzantine reliable broadcast algorithm.

INDEX TERMS Byzantine network, dynamic system, broadcast, reconfiguration.

I. INTRODUCTION

Replicating service state is a common technique to design a reliable distributed system. However, in long-lived systems, a typical fault-tolerant protocol may be inadequate, because the possibility of losing more than a minority of participants unavoidably grows with time. In addition, the need to react to changes in application workloads requires the system to dynamically add resources or replace old and slow machines with newer and faster ones. Real-world distributed systems need to be dynamic, i.e., they need to update the active members of a system over time. Such a churn support can be realized by providing interfaces to *reconfiguration* operations, i.e., *add* or *remove* participants.

It is already a challenge to design fault-tolerant distributed services that provide strong availability and consistency. Further requirements on supporting updating participants dynamically complicate any system design.

The first challenge in reconfiguration problems is safety when concurrent reconfiguration requests are submitted to the system. Consider an example from [1] where we build

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

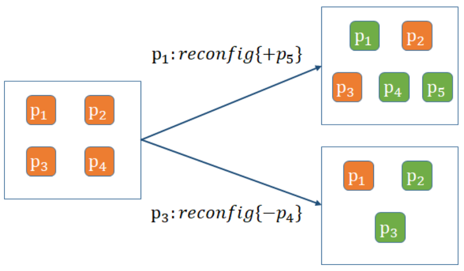

FIGURE 1. Concurrent reconfigurations in systems may lead to inconsistency.

a fault-tolerant solution using four nodes p_1, p_2, p_3, and p_4. As shown in Figure 1, a user adds a new node p_5 to the system through p_1 (i.e. by issuing a adding command at node p_1). The up-to-date system configuration is stored at a majority of nodes of the current configuration, e.g., at nodes p_1, p_4 and p_5. At the same time, the removal of p_4 is initiated by a reconfiguration operation at p_3. If p_2 and p_3 do not know that node p_5 should be added to the system, they consider themselves the majority of the latest configuration in the system. When these two concurrent reconfigurations occur, the system diverges to two different configurations. Early work on system dynamics could violate safety in such cases [2], [3], [4]. This problem has been solved in many subsequent works by establishing a certain update order [5], [6], [7]. This is equivalent to achieving an agreement in the system or assuming an external, replicated configuration manager [8], [9].

Second, the system's liveness is in danger if arbitrary reconfigurations are allowed. Consider an example from [10]: Three nodes, p_1, p_2, and p_3 are in a fault-tolerant system. The latest system state is stored with a majority of nodes of the current configuration, e.g. p_1 and p_2. If a user removes node p_1, node p_2 becomes absolutely vital, and cannot crash anymore. Otherwise, the remaining nodes cannot keep the correct system state. This problem will be even more severe if we introduce Byzantine failures to the system. Therefore, it is necessary to regularize a general characterization of tolerable failures for protecting a system's liveness.

As we know, it has been proved that achieving agreement is impossible in a fully asynchronous message-passing distributed system in the existence of even one faulty node [11]. Recent works [10], [12], [13], [14], [15] focus on dynamic distributed systems without consensus. Most of previous reconfiguration results in asynchronous systems without consensus assume that nodes always behave correctly. However, in reality, malicious nodes may arbitrarily attack the system. Thus, the system must be robust against Byzantine nodes.

In this paper, we study the dynamic problem in asynchronous systems with Byzantine faults. We are given an asynchronous message-passing distributed system composed of a set $\prod = \{p, q, \dots\}$ of interconnected nodes. We want to design protocols that provide interfaces to *reconfiguration* operations with existence of Byzantine nodes.

In summary, this paper features in following contributions.

- We define a dynamic Byzantine broadcast problem including an explicit liveness condition, which is independent of a particular solution to the problem.
- We solve the dynamic Byzantine problem by designing a dynamic Byzantine consistent broadcast protocol and a dynamic Byzantine reliable broadcast protocol in a completely asynchronous message-passing system.

We follow the framework of dynamic reconfiguration without consensus in the crash-prone model and adjust it to Byzantine failures. Our algorithms realize consensus-less reconfiguration operations with Byzantine failures.

Compared to other dynamic broadcast protocols [16], [17], [18], [19], [20], our results do not require a completely ordered sequence of configurations, or equivalent agreement among nodes. Moreover, our design can be implemented in fully asynchronous systems. Also, our protocols are highly modular, and not restricted to any specific broadcast primitives. In this paper, we present two different broadcast protocols and discuss how to extend our protocols to further models.

The rest of this paper is organized as follows: In the next section, we give an overview of related work. Section 3 defines the model that we study in this paper. In Section 4, we discuss the dynamic problem with Byzantine failures. Section 5 introduces the weak snapshot object, which provides fundamental functions for constructing our broadcast protocols. In Section 6, we present two dynamic Byzantine broadcast protocols. We conclude this paper in Section 7.

II. RELATED WORK

In this section, we first review several existing solutions retrospectively as solving a dynamic problem. Then, we move to three specific topics that consider broadcast services with dynamism.

The majority of related work on reconfiguration considers read/write storage. RAMBO [5], [8] was the first to implement a dynamic atomic register with asynchronous read/write operations. The main idea of the protocol is to use consensus to agree on reconfigurations. It supports operations to check all available configurations in the system and write to the latest one. Other works [9], [21], [22] use the same idea to design dynamic atomic registers with Byzantine failures.

Other approaches provide a fault-tolerant emulation of arbitrary data types with the State Machine Replication (SMR) [23], [24]. The first method is to achieve an agreement on a sequence of operations applied to the data. SMR was implemented in Paxos [23], which allows nodes to change the system configuration by keeping the configuration as part of the state stored in the state machine. Another method is to introduce a secondary configuration manager to arrange configuration changes. The manager engages this information into the replication protocol. Several practical systems [19], [25], [26], [27] use this method to achieve dynamism.

Another study [28] considers the situation that the opportunity of losing more than a minority of participants unavoidably grows. They design a failure detector to deal with the number of failures right from the beginning. However, their model does not allow adding more participants to the system.

Unlike other previous studies, DynaStore [10] was the first to the solve dynamic reconfiguration problem in asynchronous message-passing systems without consensus. It uses a weak snapshot abstraction to formulate a digraph containing information about the changes of system configurations. Even though it does not solve the agreement problem in asynchronous systems, it still ensures the safety by traversing the digraph of the system configuration. Followed by that, SmartMerge [14] and Parsimonious SpSn [13] use very similar ideas to solve the dynamic problem. We also employ the similar idea to construct a digraph of system configurations with the weaker snapshot abstraction in this paper. Moreover, we extend this concept for Byzantine fault tolerance.

Dynamic Broadcast protocols have been studied even earlier than the reconfigurable read/write storage. In 1987, a protocol [17] was proposed for failure-free scenarios. Several works [1], [22], [24], [27] extend this idea for fault tolerance, where failure-prone nodes are replaced by fault-tolerant disjoint groups.

Configuration-oriented group communication systems maintain a dynamic configuration of active participants. They provide broadcast and atomic multicast services within the member of a configuration by keeping agreement on a sequence of configurations [29].

Other dynamic broadcast protocols [16], [30] do not solve the reconfiguration problem. Instead, they seek to allow replicas to dynamically change the broadcast groups they subscribe to, while the membership of the system remains constant.

Guettaoui *et al.* [31] addressed a similar problem as we study in this paper. Compared to algorithms that we propose, the previous work only provides interfaces for broadcasting a single message. However, the broadcast abstractions [32] for Byzantine failures in a static configuration always support arbitrary number of messages, which is a practical regiment in reality. Our protocols support broadcasting arbitrary number of messages in dynamic configurations, which can be considered as broadcast channels in the system.

Moreover, our protocols are not restricted to any specific broadcast abstractions, it can be easily adjusted to other Byzantine broadcast primitives in distributed applications. In this paper, we present a consistent broadcast and discuss how to modify it for other broadcast abstractions.

III. DISTRIBUTED SYSTEM MODEL

Consider a distributed system composed of a set $\prod = \{p, q, \ldots\}$ of interconnected nodes. Some nodes in the system are Byzantine. They may behave arbitrarily, i.e., generate fake messages, drop messages, or even corrupt messages.

Any nodes pair u and v can send messages to each other directly. The transmission mechanism is point-to-point, i.e., nodes send a message to at most one neighbor at a time. The communication channels between nodes are authenticated. Nodes can recognize who is the sender of the message when they receive a message. Messages cannot be modified by any third parties if messages are delivered via the authenticated channel between nodes.

The system we study in this paper is asynchronous. In asynchronous distributed systems, the message delay from one node to another is without finite upper bound, i.e. messages may be delayed for arbitrary time periods. However, messages will eventually be delivered. Nodes only take actions when they are activated by events, such as messages arriving.

We assume that the total number of nodes are unbounded and possibly infinite. However, not every node remains active in the system throughout the life of the system. At the beginning of execution time, there is a set of available nodes, *Init*, which can be accessed by users. Other nodes are initially inactive. Active set *Init* is known to every node. Nodes can be activated when joining the system or be removed from the system during execution. Once they have been removed from the system, they cannot become active again.

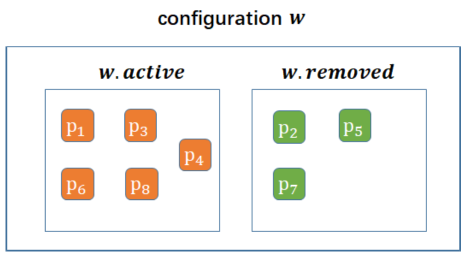

FIGURE 2. Configuration w of a system with 5 active nodes and 3 removed nodes.

Algorithms we propose in this paper use *operations*, *functions* and *upon clauses*. Operations are invoked by users. Functions are invoked by operations, upon clauses, or other functions. When a node p receives messages from other nodes, it stores these messages in a local buffer. The *upon clauses* are internal actions enabled when some conditions hold. In the face of concurrency, at most one operation and one upon clause can be performed simultaneously. Note that operations and upon clauses might execute concurrently.

IV. PROBLEM DEFINITION

The goal of our work is to implement a dynamic broadcast service which tolerates byzantine failures. The broadcast service is deployed on a collection of nodes that interact with asynchronous message-passing.

A. BYZANTINE DYNAMIC BASICS

We give a formal definition of a system configuration below, which comes from [10]. An example of a system configuration w is presented in Figure 2.

Definition 1 (Configuration): A configuration w represents sets of active nodes and removed nodes. It holds two sets: The first set is a set of active nodes, $w.active$. Nodes in $w.active$ are either active at the beginning of the system or are activated during the execution but not removed from the system. The second set is a set of removed nodes $w.removed$, which are nodes removed from the system. The size of w is the sum of the size of $w.active$ and the size of $w.removed$.

Users can change the system configuration by reconfiguration operations. A set of configuration changes cng contains two subsets. Subset $cng.join$ contains nodes which are activated for joining the system and subset $cng.remove$ contains nodes which will be removed from the system.

We define an addition operation between a configuration w and a set of changes cng. The result of $w' = w + cng$ is a configuration. $w'.active = w.active \cup cng.join \setminus cng.remove$ and $w'.removed = w.removed \cup cng.remove$. An example of such computation is shown in Figure 3.

Once a node is excluded from the system, it can no longer be included in the system with the same identity. If there exists some change sets $cng_1, cng_2, \ldots$ that validate the equation $w + cng_1 + cng_2 + \ldots = w'$, then we have $w \subseteq w'$.

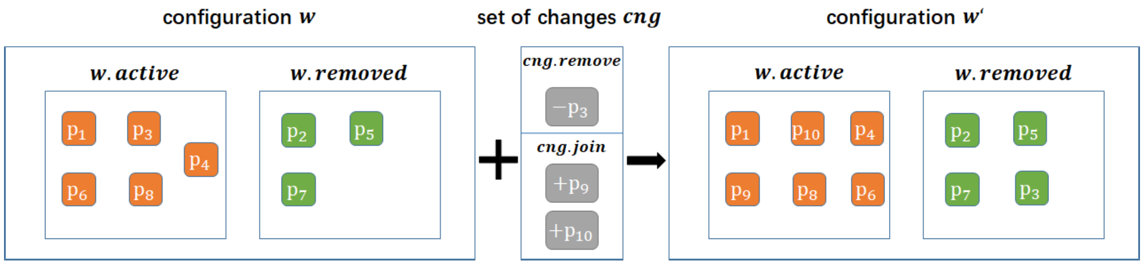

FIGURE 3. Applying a change set *cng* to a system configuration *w*. The new configuration *w′* contains 6 members and 4 nodes have been removed from the system.

B. LIVENESS CONDITIONS

Before we specify the liveness conditions for the dynamic reconfiguration problem with Byzantine failures, we first indicate some preliminary definitions for this problem.

At any time t in the execution, we define $P(t)$ be the set of pending changes at time t such that a $reconfig(cng)$ was invoked but has not completed. We define $B(t)$ as the set of nodes that are Byzantine (including crashed nodes) by time t.

It is impossible to achieve agreement in an asynchronous message passing system with Byzantine failures [11]. Therefore, the configurations known by each node may be different during the execution. Let us denote $CurConfig_p(t)$ as the latest configuration that node p keeps at time t. A configuration $AvaiConfig$ is available at time t if and only if there exists an active node p such that $CurConfig_p(t) \subseteq w$.

The following liveness conditions hold during the whole execution time of the system: at any time t, for any available configuration $AvaiConfig$ in the system at time t, fewer than $\frac{|AvaiConfig.active|}{3}$ nodes out of set $P(t).join \cup AvaiConfig.active$ are in set $B(t) \cup P(t).remove$.

C. DYNAMIC BROADCAST WITH BYZANTINE FAULTS

The dynamic broadcast is a communication abstraction that helps with highly available and scalable system applications. Dynamic broadcasts support two communication primitives, $broadcast(m)$ and $deliver(m)$, and a reconfiguration primitive, $reconfig(cng)$. In this paper, we realize dynamic Byzantine broadcast primitives, which allow nodes to broadcast an arbitrary number of messages by invoking $broadcast(m)$ multiple times and enable communications beyond point-to-point in the system.

Users call $broadcast_p(m)$ to broadcast a message m in the system. When a message m is delivered at node p, node p will get an indication such that a message m with label l broadcast by node p is delivered. Label l can be an arbitrary bit string. All messages delivered for a particular sender are distinctively labeled.

In addition to *broadcast* and *deliver* primitives, the service exposes an interface, i.e., $reconfig_p(cng)$, for invoking reconfigurations. Operation $reconfig_p(cng)$ allows an active node p to add new nodes to the system or remove nodes from the system, where cng is a set of nodes changing.

Only nodes that are members of the current system configuration should be allowed to initiate actions. Therefore, the $broadcast_p$, $reconfig_p$ operations at node p are initially disabled, until an *enable* event occurs at p. Operations are no longer accessible at node p if $halt_p$ event is invoked. After terminated by the $halt_p$ event, node p can still deliver messages that were broadcast before it left.

We assume that Byzantine nodes cannot arbitrarily remove or add nodes. Otherwise, the system will not be valid anymore if infinite Byzantine nodes are added to the system. The correct reconfiguration behavior can be regulated outside our protocols. We say node p is active in the system if and only if it is added to the system correctly, i.e., it starts with a correct configuration.

We study two dynamic Byzantine broadcast protocols in this paper: dynamic Byzantine consistent broadcast and dynamic Byzantine reliable broadcast. Both of them support multiple messages broadcast. We point out the properties of these two broadcasts as follows.

The dynamic Byzantine consistent broadcast has four properties [33]:

Validity: If a correct node (active node that follows the algorithm protocol) p broadcasts a message m, then every correct node q which joins the system before and does not leave until after the broadcast of m eventually delivers m.

No duplication: If a correct node p broadcasts a message m with label l, then every other correct node delivers at most one message with label l from p.

Integrity: If a correct node q delivers a message m from sender p and p is correct, then m was previously broadcast by p.

Consistency: If a correct node q delivers a message m with label l from sender p, and another correct node s delivers a message m' with label l from sender p, then $m = m'$.

The dynamic Byzantine reliable broadcast has four properties [33]:

Validity, **No duplication**, **Integrity:** same properties as the dynamic Byzantine consistent broadcast.

Agreement: If a correct node q delivers a message m with label l from sender p in configuration w, then every other correct node in configuration w delivers a message m with label l from sender p eventually.

V. THE WEAK SNAPSHOT ABSTRACTION

Before discussing broadcast protocols, we first introduce a weak snapshot object [10] S, which provides fundamental functions for constructing broadcast protocols.

A weak snapshot object ws is accessible by a set P of nodes. It supports two operations for every node $p \in P$: $update_p(cng)$ and $scan_p()$. Operation $update_p(cng)$ has an input value cng. The operation returns a Boolean value that indicates whether the update operation success. Operation $scan_p()$ returns a set of values that have been successfully updated by $update_p()$ previously. For each weak snapshot object, the nodes set P is static. Operations $scan_p()$ and $update_p()$ have following properties:

integrity: Let o be a $scan_p()$ operation that returns C. Then, for each $cng \in C$, an $update(cng)$ operation is invoked by some node q prior to the completion of o.

validity: Let o be a $scan_p()$ operation that is invoked after the completion of an $update(cng)$ operation, and that returns C. Then, $C \neq \emptyset$.

monotonicity of scans: Let o be a $scan_p()$ operation that returns C and let o' be a $scan_q()$ operation that returns C'. If o' is invoked after the completion of o. Then, $C \subseteq C'$.

non-empty intersection: There exists a change set cng such that for every $scan()$ operation that returns $C \neq \emptyset$, it holds that $cng \in C$.

termination: If some majority M of nodes in P do not crash, then every $scan_p()$ and $update_p(cng)$ invoked by any node $p \in M$ eventually completes.

Generally speaking, these properties require that all *scan* operations with non-empty returns observe the "first" *update* operation.

Every configuration w in the system has a weak snapshot object $ws(w)$, which can be accessed by nodes in $w.active$. The algorithm uses an array of single-writer multi-reader atomic registers as $Mem[w]$ of size $|w.active|$ for each weak snapshot object. We use the Byzantine Quorum with Listeners algorithm [34] to implement such Byzantine atomic registers in asynchronous message-passing systems.

Operation $collect_p()$ reads updates which are stored in $Mem[w]$. Node p invokes $update_p(w, cng)$ to write a set of changes cng to the $Mem[w][p]$ in $ws(w)$, which is considered as reconfigurations to configuration w. In $update_p(w, cng)$, the algorithm first reads all updates that have been written in $Mem[w]$ by invoking $collect(w)$. If it has not found any existing updates, it writes cng in $Mem[w][p]$. Change set cng will be observed by following $collect_q(w)$ functions for any $q \in w.active$. Node p learns updates proposed by other nodes to w by invoking $scan_p(w)$. It calls $collect(w)$ to read all previous updates to w. If no update has been written in $Mem[w]$, it returns an empty set $\emptyset$. Otherwise, it calls $collect(w)$ again and returns all previous updates.

The relationship between configurations in a system can be represented by a digraph $G = (V, E)$. Each vertex in G indicates a system configuration. There is a directed edge pointing from vertex v_w to vertex $v_{w'}$ if and only if the following relationship holds for the configuration w and the configuration w', i.e., $w' = w + scan_p(w)$ for a node p.

Algorithm 1 Weak Snapshot-Code for p

```
function $update_p(w, cng)$
    if $collect() = \emptyset$ then
        $Mem[w][p].Write(cng)$
        return $true$
    else
        return $false$
    end if
end function
function $scan_p(w)$
    $C \leftarrow collect(w)$
    if $C = \emptyset$ then
        return $\emptyset$
    end if
    $C \leftarrow collect(w)$
    return $C$
end function
function $collect_p(w)$
    $C \leftarrow \emptyset$
    for all $q \in w.active$ do
        $cng \leftarrow Mem[w][q].Read()$
        if $c \neq \perp$ then
            $C \leftarrow C \cup \{cng\}$
        end if
    end for
    return $C$
end function
```

Intuitively, we use these weak snapshots as follows: when a node p wants to change the configuration w, it invokes $update_p(w, cng)$ to propose a set of changes cng; it learns the changes of configuration proposed by other nodes to configuration w by invoking $scan_p(w)$, which returns a set of changes. Compare to atomic snapshot objects [35], we do not require atomicity, i.e., all operations can be ordered in a sequential execution.

Integrity and termination properties have been guaranteed by the correctness of Byzantine atomic registers [34]. Aguilera *et al.* [10] explained in detail why Algorithm 1 preserves validity, monotonicity and non-empty intersection properties.

VI. DYNAMIC BYZANTINE BROADCAST

This section describes in detail the dynamic Byzantine broadcast protocols. We first introduce the consistent broadcast from Section 6.1 to Section 6.4. Then we present the reliable broadcast in Section 6.5.

Key components of our consistent broadcast protocol are shown in Algorithm 3, which contain function *BroadcastInConfig* (lines 48-56) and *upon clauses* (lines 73-86) for broadcasting and delivering messages in a given configuration. We realize the broadcast service in a given configuration with the authenticated echo broadcast

Algorithm 2 Code for Node p, Local Variables

```
1: local variables
2:     $MsgNum_p$, initially 0,          ▷ the number of messages that $p$ broadcasts to others
3:     $ConfigList_p$, initially ∅,       ▷ the set of configurations that visited by node $p$
4:     $CurConfig_p$, initially $Init$,   ▷ latest configuration found by $p$ in the system
5:     $ReceivedM_p$, initially ∅,        ▷ the set of $ECHO$ messages received by node $p$
6:     $ValidatedM_p$, initially ∅,  ▷ the set of messages validated by a quorum of a configuration, waiting for final delivery by
   node $p$
7:     $DeliveredM_p$, initially ∅,       ▷ the set of messages delivered by node $p$
8: end local variables
```

protocol [36], [37]. The algorithm has two phases, broadcast-phase and check-phase. When there are no reconfigurations, all nodes stay in the same configuration. Operation *broadcast* executes a check-phase to find configuration updates in the system and enters the broadcast-phase after the check-phase. When a node receives new messages from other nodes, it runs a check-phase to update its current configuration.

To allow reconfiguration, system members store information about configuration changes in weak snapshot objects. Members of a configuration are able to change the configuration by updating a change set in the weak snapshot object. We allow concurrent *reconfiguration* operations with any *broadcast* operations and *deliver* events. Node p delivers all messages that broadcast between its appearance in the system and its departure. Furthermore, once a *reconfiguration* operation is completed, future *broadcast* can be carried out in the new configuration. The key challenge is to ensure that no message delivers linger behind in the old configuration, while updates are made to the new configuration. We use the following strategies to make sure the correctness of broadcast services in dynamic systems.

- In our dynamic protocol, nodes do not directly deliver messages when it is confirmed by a quorum of the system. The procedure is modified so that nodes first accept all messages that they receive in any configurations. Then nodes deliver messages according to the configuration they have observed. Messages are successfully delivered at node p if they are broadcast in a configuration that p has visited. If a new message is sent to a node, the node starts the check-phase to detect new configurations in the system.
- The broadcast-phase works in the latest configuration found by the check-phase. First, nodes broadcast the message in the latest configuration during the broadcast-phase, and then nodes read the reconfiguration information. If nodes find a new configuration, the protocol restarts the check-phase.
- The *reconfig* operation starts with a check-phase and then writes information about the new configuration to the quorum of the old configuration. These procedures will be iterated until no new configurations are found.

Function *BroadcastInConfig* is the core of a broadcast phase, which implements the basic functionality of the broadcast-phase. It first broadcasts the message in the current configuration (line 50-52), and then finds previous update of the current configuration by scanning its weak snapshot object in line 54. To deliver a message, nodes run function *ReceiveMsg* when they receive messages from other nodes (line 84) or visit a new configuration (line 26).

Let us discuss a simple example where only one reconfiguration request, i.e., RC, is called, from w_1 to w_2. Consider a *broadcast* operation, i.e., B, runs a broadcast-phase with message m. There are two possible cases with respect to B. One is that it does not observe any updates of configurations. This means that B's execution of broadcast-phase is in w_1 and all correct nodes will receive m and deliver it successfully. Otherwise, B observes RC in its check-phase and broadcasts m in w_2. After receiving messages from other nodes, correct nodes start check-phase to detect new configuration in the system. Because RC is already found by B, it will be also found by other check-phases which start after B's check-phase. Message m will be delivered in w_2 at correct nodes eventually.

In the example above, we ignored the correctness of the dynamic broadcast if several nodes concurrently propose changes to w_1. Therefore, the rest of our algorithm aims to handle the complexity that incurs due to multiple reconfiguration requests. In Section 6.1, we introduce the local state of nodes. Section 6.2 presents the pseudo-code of the dynamic Byzantine consistence broadcast. We discuss the notion of established configurations in Section 6.3. The analysis and proof of the correctness are presented in Section 6.4. Finally, we further introduce the dynamic Byzantine reliable broadcast in Section 6.5.

A. STATE OF NODES

In this subsection, we discuss the local state of nodes. Each node stores several variables in local memory. These variables are shown in Algorithm 2. Note that all actions to local variables are atomic. Local variables at node p are denoted with subscript p.

All configurations which have been visited by node p are stored in $ConfigList_p$, which is initialed as an empty set. The latest configuration found by node p is kept in $CurConfig_p$, which is initialed as $Init$, i.e., the primary system configuration. $MsgNum_p$ is the number of messages that broadcast by node p.

Algorithm 3 Code for Node p, Dynamic Byzantine Consistent Broadcast

```
1: initially:
2:     if $p \in Init.active$ then
3:         enable operations
4:     end if
5: end initially:
6: operation $broadcast_p(m)$
7:     $newConfig \leftarrow Traverse_p(\emptyset, m)$
8:     $MsgNum_p \leftarrow MsgNum_p + 1$
9: end operation
10: operation $reconfig_p(cng)$
11:     $CurConfig_p \leftarrow Traverse_p(cng, \perp)$
12:     for all $q \in cng.join$ do
13:         send $(Join, newConfig, p)$
14:     end for
15:     return $newConfig$
16: end operation
17: function $Traverse_p(cng, m)$
18:     $desiredConfig \leftarrow CurConfig_p + cng$
19:     $Front \leftarrow \{CurConfig_p\}$
20:     loop
21:         $w \leftarrow l \in Front$ s.t. $l$ is the minimal configuration in $Front$
22:         if $p \notin w.active$ then
23:             $halt_p$
24:         end if
25:         $ConfigList_p \leftarrow ConfigList_p \cup w$
26:         $ReceiveMsg_p()$
27:         if $w \neq desiredConfig$ then
28:             $update_p(w, desiredConfig \setminus w)$
29:         end if
30:         $ChangeSets \leftarrow scan_p(w)$
31:         if $ChangeSets \neq \emptyset$ then
32:             $Front \leftarrow Front \setminus \{w\}$
33:             for all $c \in Changesets$ do
34:                 $desiredConfig \leftarrow desiredConfig + c$
35:                 $Front \leftarrow Front \cup \{w + c\}$
36:             end for
37:         else
38:             $ChangeSets \leftarrow BroadcastInConfig_p(w, m)$
39:             $m \leftarrow \perp$
40:         end if
41:         if $ChangeSets = \emptyset$ then
42:             $CurConfig_p \leftarrow desiredConfig$
43:             return $desiredConfig$
44:         end if
45:     end loop
46: end function
47: function $BroadcastInConfig_p(w, m)$
48:     if $m \neq \perp$ then
49:         for all $q \in w.active$ do
50:             send (SEND, $p$, $MsgNum_p$, $m$, $w$) to $q$
51:         end for
52:     end if
53:     $ChangeSets \leftarrow scan_p(w)$
54:     return $ChangeSets$
55: end function
56: function $ReceiveMsg_p$
57:     for all $q \in CurConfig_p.active \bigcup CurConfig_p.removed$ do
58:         for all $(q, MsgNum_q, m, w) \in ValidatedM_p$ do
59:             if $(q, MsgNum_q, m, *) \notin DeliveredM_p$ and $w \in ConfigList_p$ then
60:                 $DeliveredM_p \leftarrow DeliveredM_p \cup (q, MsgNum_q, m, w)$
61:                 $ValidatedM_p \leftarrow ValidatedM_p \setminus (q, MsgNum_q, m, w)$
62:             end if
63:         end for
64:     end for
65: end function
66: upon receiving (SEND, $s$, $MsgNum$, $m$, $w$) from node $s$
67:     for all $q \in w.active \setminus s$ do
68:         send (ECHO, $p$, $s$, $MsgNum$, $m$, $w$) to $q$
69:     end for
70: end upon
71: upon receiving (ECHO, $q$, $s$, $MsgNum$, $m$, $w$) from node $q$
72:     $ReceivedM_p \leftarrow ReceivedM_p \cup (q, s, MsgNum, m, w)$
73:     if $\#t > \frac{2 \cdot |w.active|}{3}$
74:       $s.t.(t, s, MsgNum, m, w) \in ReceivedM_p$ then
75:         $ValidatedM_p \leftarrow ValidatedM_p \cup (s, MsgNum, m, w)$
76:     end if
77:     $CurConfig_p \leftarrow Traverse_p(\emptyset, \perp)$
78:     $ReceiveMsg_p()$
79: end upon
80: upon receiving $(Join, w, q)$ from node $q$
81:     if $p \in w.active$ and $p \notin Init.active$ then
82:         enable operations
83:         if $CurConfig_p = Init$ then
84:             $CurConfig_p \leftarrow w$
85:         end if
86:     end if
87: end upon
```

Node p delivers a message in four steps. First, it sends *ECHO* messages to other nodes when it receives a *SEND* message from the sender. Second, it puts messages in $ReceivedM_p$ whenever it receives *ECHO* messages from others. A message m is considered to be correct in configuration w if a quorum of $w.active$ confirms its correctness with an *ECHO* message. These messages are stored it in $ValidatedM_p$ with its broadcast configuration w in the third step. Until w is visited by p, messages in $ValidatedM_p$ with configuration w are successfully delivered and enter in $DeliveredM_p$.

B. DYNAMIC BYZANTINE CONSISTENT BROADCAST

This subsection describes the algorithm for the dynamic Byzantine consistent broadcast, which is shown in Algorithm 3. Operations and functions at node p are denoted with subscript p. There are two operations that can be invoked directly by users at node p, $broadcast_p(m)$ and $reconfig_p(cng)$.

Users call $broadcast_p(m)$ to broadcast a message m in the latest configuration and propose a set of configuration changes cng to the system by invoking $reconfig_p(cng)$. We introduce these two operations respectively in this subsection. Then, we discuss how nodes deliver messages. The main logic function $Traverse_p$ is introduced in the last part of this subsection.

1) BROADCASTING MESSAGES

Operation $broadcast_p(m)$ invokes $Traverse_p(\emptyset, m)$ to find the latest configuration w in the system and broadcast message m, where $\emptyset$ implies that this execution does not change the system configuration. After broadcasting, variable $MsgNum_p$ increases by 1 for further broadcasting.

2) RECONFIGURATIONS

Similar to a $broadcast_p(m)$ operation, a $reconfig_p(cng)$ also invokes $Traverse_p(cng, \perp)$ to find the latest configuration w, and apply a change set cng to the system configuration, where $\perp$ implies that this execution does not broadcast any messages.

Node p sends the new configuration to all joining nodes in line 12-14. A new node successfully joins the system if it receives a correct configuration.

3) DELIVERING MESSAGES

Message delivery is a passive procedure in the protocol. It is triggered when nodes receive messages from other nodes. We implement an authenticated echo broadcast algorithm in a certain configuration. After receiving a message m with a $SEND$ tag broadcast in configuration w by node s (line 50-52), node p broadcasts m with $ECHO$ tag to other nodes in configuration w (line 74-76). When node p receives more than $\frac{2\cdot|w.active|}{3}$ $ECHO$ messages containing the same message m broadcast by node s, in the same configuration m, with the same message number $MsgNum$, it validates m and store it in set $ValidatedM_p$ (line 80-83). Node p starts to find the latest configuration in the system whenever it receives $ECHO$ messages from other nodes by invoking $Traverse_p(\emptyset, \perp)$ (line 84), where $\emptyset$ implies that this execution does not change the system configuration and $\perp$ implies that this execution does not broadcast any messages.

Message m received by p in configuration w is not delivered immediately because w could be a fake configuration and message m may not be broadcast in the real system. Node p delivers message m if it has visited the configuration w where m was broadcast (line 57-72). Node p checks all validated messages by calling function $ReceiveMsg_p()$ when it visits a new configuration (line 26) or it receives an $ECHO$ message (line 84).

4) TRAVERSING THE GRAPH OF CONFIGS

Weak snapshots form all configurations into a digraph, where configurations are vertices in the graph. If there is an $update_q(w, cng)$ happening with return *true* during the execution by node $q \in w.active$ where $w' = w + cng$, $cng \neq \emptyset$, we have a direct edge from vertex w to vertex w' in the digraph. Our algorithm requires that a removed node cannot join the system again, thus the size of w' is always larger than w and the graph or configurations is acyclic.

Node p gets the latest configuration information through the function $Traverse_p()$, which is invoked by all operations or when node p receives messages from other nodes. This function traverses the digraph of configurations from its current configuration $CurConfig$. Then, the digraph is visited for collecting configuration changes which are previously applied to the system. All changes are gathered in the set $desiredConfig$. After finding all changes, $desiredConfig$ is included in the digraph if it does not exist. A weak snapshot object of $desiredConfig$ is connected to the graph by function $update_p()$. Other nodes in the system can find $desiredConfig$ in subsequent searches.

We implement a well-known Dijsktra algorithm [38] to traverse the graph, with a modification such that the traversal also reforms the digraph, which is similar to the traversal algorithm in [10]. Function $Traverse_p()$ keeps a set of configurations $Front$, which contains discovered configurations waiting for further visit. Set $Front$ is initialed as $\{CurConfig_p\}$ (line 19). Each iteration, $Traverse_p$ visits the nearest configuration from $CurConfig_p$ in $Front$ (line 21), i.e., the configuration which has the minimal size in the set, and remove this vertex from $Front$ (line 33) after visiting. If more than one configuration has the same size, $Traverse_p()$ first visits the one with the minimal size of *removed* subset.

When $Traverse_p()$ visits a configuration w (line 20-46), it first checks if p is still in the system and then delivers messages that p has stored in $ValidatedM_p$ by invoking $ReceuveMsg_p()$. It is possible that w does not include all known changes found by the traversal, i.e., $w \neq desiredConfig$. In this case, p calls $update_p()$ to build an edge from w to $desiredConfig$ in the digraph (line 28-30). This update can be failed because another node has already built an edge outgoing from w. If so, $Traverse_p()$ will visit the successor node of w and update it again.

Once an update has been applied to the weak snapshot of w successfully (line 29), latter $scan_p(w)$ (line 31) will return the update. A non-empty changes set is returning if there are edges outgoing from w. For every change set $c \in ChangeSets$ returned by $scan_p(w)$, there is an edge from w to $w + c$. Thus, we add $w + c$ to $Front$ for furthering visit

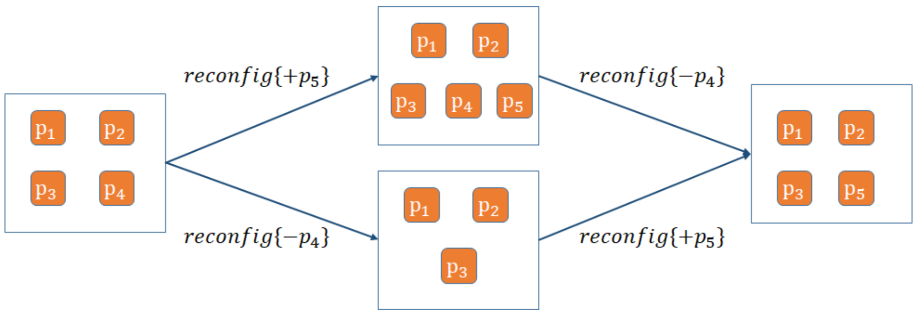

FIGURE 4. Traversal of the digraph of configurations.

(line 36). Change sets c is also collected by *desiredConfig* (line 35). If there is no outgoing edges from w, node p invokes $BroadcastInConfig(w, m)$ (line 39), which broadcasts the message in w (line 50-52) and scans the weak snapshot object of w again (line 54). If there is nothing found by $scan_p(w)$, function $Traverse_p()$ completes.

When there is only one configuration in *Front*, *desiredConfig* is visited by $Traverse_p()$ because it has the maximum size among all configurations visited by $Traverse_p()$ in the execution. This configuration contains all changes observed during the traversal. Moreover, if $w \neq desiredConfig$, the condition in line 32 is always satisfied. Therefore, $BroadcastInConfig(w, m)$ is invoked when there is only one configuration in *Front*, i.e., *desiredConfig*. For the same reason, when $Traverse_p$ finishes, we have $Front = \{desiredConfig\}$. This configuration is assigned to $CurConfig_p$ in line 43 and returned by $Traverse_p$.

We consider the example that we discussed in Introduction. Operations $reconfig_p(cng_1)$ and $reconfig_q(cng_2)$ are executed concurrently, where $cng_1 = \{+p_5\}$ and cng_2 $\{-p_4\}$. There are two possible cases resulted from such simultaneous operation. In the first case, either cng_1 or cng_2 are first updated succefully with a $update(w, cng)$ function. Suppose it is cng_1. The other update function observes cng_1 and will not write cng_2 in the weak snapshot object of w. Function $Traverse_q()$ invoked by $reconfig_q(cng_2)$ renews its *desiredConfig* from $w+cng_2$ to $w+cng_2+cng_1$, visits configuration $(w+cng_1)$ in the next iteration and writes cng_2 in $ws(w+cng_1)$. $Traverse_p()$ will find $w+cng_2+cng_1$ from $w+cng_1$ in future executions.

In the second case, both cng_1 and cng_2 are successfully written into the weak snapshot object of w. Benefited from the validity property of the weak snapshot object, $scan_p(w)$ and $scan_q(w)$ both return non-empty sets C_p and C_q. The non-empty intersection property of the weak snapshot object ensures $C_p \cap C_q \neq \emptyset$, i.e., either $C_p \cap C_q = cng_1$, $C_p \cap C_q = cng_2$ or $C_p \cap C_q = cng_1 \cup cng_2$. Suppose it is cng_1. Then $Traverse_q()$ discovers both configuration $w + cng_1$ and configuration $w + cng_2$. It will visit them in the following iterations and build edges from configuration $w + cng_1$ and configuration $w + cng_2$ to configuration $w + cng_1 + cng_2$. $Traverse_p()$ will find $w + cng_2 + cng_1$ in future executions. If $C_p \cap C_q = cng_1 \cup cng_2$, both $Traverse_p()$ and $Traverse_q()$ will find $w + cng_1 + cng_2$ in this execution. The digraph of this example is shown in Figure 4.

C. SEQUENCE OF ESTABLISHED CONFIGURATIONS

The difficulty of the reconfiguration problem is that different nodes might keep different configurations if several reconfiguration operations are invoked concurrently. In our algorithm, when multiple nodes invoke $update(w)$ at the same time, concurrent $scan(w)$ might see different outgoing edges from w.

Fortunately, the non-empty intersection property of weak snapshot objects ensures that nodes will never work on different branches of the digraph. Actually, at least one outgoing edge is returned by all $scan(w)$ functions and the destination of this edge will be visited by all $traverse()$ functions. This fact enables us to define a totally ordered subset of configurations, i.e., *established* configuration, as follows:

Definition 2 (Sequence of Established Configurations [10]): The unique sequence of established configurations ε is constructed as follows:

- *the first configuration in ε is the initial configuration Init;*
- *if w is in ε, then the next configuration after w in ε is $w' = w+cng$, where cng is an element chosen arbitrarily from the intersection of all sets $C \neq \emptyset$ returned by some $scan(w)$ operations in the execution.*

The first digraph traverse in the system starts from $CurConfig_p = Init$, which is an established configuration by definition. At each iteration, $Traverse_p()$ visits a configuration w, removes it from *Front* and adds its children in *Front*. If w is an established configuration, then one of its children is also an established configuration, which is included in *Front*. If w is not established, an established configuration is still in *Front*. Thus, at least one established configuration in *Front* in each iteration. At the end of the execution of $Traverse_p$, there is only one configuration *desiredConfig* in *Front*. Therefore, configuration *desiredConfig* is also an established configuration. We conclude that *desiredConfig* which is assigned to $CurConfig_p$ in line 43 and returned from $Traverse_p$ is established at the first digraph traverse. By induction, $Traverse_p()$ always starts with an established configuration and returns an established configuration. Configuration $CurConfig_p$ is also always established.

Function $BroadcastInConfig_p$ (line 48-56) is performed in an established configuration as well. Moreover, we prove that each $Traverse_p()$ visits every established configuration in ε between the starting configuration $CurConfig_p$ and the returned configuration $desiredConfig$. Thus, intuitively, by visiting each configuration during a traversal, it is guaranteed to never broadcast on different branches on the digraph and all correct messages can be delivered eventually.

Lemma 1: Whenever $BroadcastInConfig_p(w, *)$ is invoked, w is an established configuration.

Proof: As we discussed above, the first digraph traversal stars from an established configuration $CurConfig_p = Init$. Therefore, by induction, configuration $desiredConfig$ returned from $Traverse_p$ in line 44 is always established and $CurConfig_p$ is always an established configuration.

We claim that if $scan_p(w)$ returns $ChangeSets = \emptyset$ at some iterations of the loop in lines 20-46, then $w = desiredConfig$ and $Front = \{desiredConfig\}$.

If w is not equal to $desiredConfig$, then there must be a function $update_p(w, desiredConfig \setminus w)$ which completes before $scan_p(w)$ starts. This is impossible because of the validity property of weak snapshot objects.

Moreover, if $Front \neq \{desiredConfig\}$, there is another $w' \in Front$. We either have $|w| < |w'|$, or $|w'.removed| < |w.removed|$. Because $w = desiredConfig = w' + c$ for some change set c, these inequalities are impossible to hold.

Since during the iteration, $scan(w)$ returns $\emptyset$, $Front$ does not change from the beginning of the iteration. Any configuration w passed to $BroadcastInConfig_p(w, *)$ is established. □

Lemma 2: Let T be an execution of $Traverse_p()$ and initConfig be the value of $CurConfig_p$ when node p starts this execution. Let desiredConfig be the value of $CurConfig_p$ when this execution terminates. Then $Traverse_p()$ visits all configurations from the sequence of established configurations between initConfig and desiredConfig.

Proof: When T begins, $Front = \{initConfig\}$.

Let $estConfig = initConfig$. Consider some iterations during T visit $w \neq estConfig$. This happens only if $estConfig$ is removed from $Front$ and some $scan(estConfig)$ during T returns a non empty $ChangeSets$.

After removing $estConfig$ from $Front$, for every $c \in ChangeSets$ returned by $scan(estConfig)$, $(estConfig + c)$ is added to $Front$. By the non-empty intersection property of weak snapshot objects and the definition of the sequence of established configurations, the configuration after $estConfig$ in ε is added to $Front$ and will be visited by $Traverse_p$ in future iterations.

Arguments above hold for every established configuration. When $BroadcastInConfig(w, *)$ is invoked, $scan(w)$ completes in line 31 and returns $\emptyset$, which ends the proof. □

D. CORRECTNESS

In this subsection, we show that our algorithm satisfies all properties of the dynamic Byzantine consistent broadcast. We indicate the correctness of the algorithm respectively in Lemma 3, Lemma 4, Lemma 5 and Lemma 6.

Lemma 3: Our algorithm satisfies the validity property.

Proof: If node q joins the system before node p broadcasts the message m, then p will enter an established configuration w where $q \in w.active \cup w.removed$. Because q does not leave the system when p broadcasts m, $q \notin w.removed$.

When node q joins the system, it starts with an established configuration which is already included in the digraph. So this configuration must be visited by node p when it broadcast m. By lemma 2, q will also visit w and deliver m eventually. □

Lemma 4: Our algorithm satisfies the no duplication property.

Proof: Because node p only broadcasts m in a certain configuration w and updates $MsgNum_p$ for each broadcasting. The $MsgNum$ and configuration form a unique label for each message. In addition, the no duplication property of the authenticated echo broadcast algorithm ensures that there is no duplication in broadcasting of messages in a certain configuration.

Therefore, every correct node only delivers at most one message with label l from sender p.

□

Lemma 5: Our algorithm satisfies the integrity property.

Proof: The integrity property of the authenticated echo broadcast algorithm ensures that if node q validates a message in a correct configuration w from sender p, it must previously broadcast by sender p in configuration w.

According to Lemma 1, a correct node p only broadcasts messages in established configuration. Hence, a correct node q delivers a message m in an established configuration after p broadcasting it.

Our assumption of the number of Byzantine nodes and the correctness of reconfiguration operations ensure that every correct node will never visit a fake configuration. Therefore, any validated messages in fake configurations will never be delivered.

□

Lemma 6: Our algorithm satisfies the consistency property.

Proof: The integrity property of the authenticated echo broadcast algorithm directly implies that if correct node q delivers a message m with label l from sender p in configuration w and correct node q delivers a message m' with label l from sender p in configuration w, then $m = m'$.

The label in our protocol contains the information of the system configuration. Therefore, if two delivered messages have the same label, then they are delivered in the same configuration. The consistency property holds intuitively. □

Theorem 1: Algorithm 3 realizes a dynamic Byzantine consistent broadcast.

Proof: Because Algorithm 3 satisfies validity, no duplication, integrity, and consistency properties, it realizes a dynamic Byzantine consistent broadcast. □

E. DYNAMIC BYZANTINE RELIABLE BROADCAST

In this subsection, we present the dynamic Byzantine reliable broadcast protocol. Only the differences between the consistent broadcast and the reliable broadcast are listed in Algorithm 4.

Algorithm 4 Code for Node p, Differences From the Consistent Broadcast Protocol

```
1:  local variables
2:      $ReadyM_p$, initially $\emptyset$,   ▷ the set of READY messages
3:  end local variables
4:  upon receiving (ECHO, $q, s, MsgNum, m, w$) from node $q$
5:      $ReceivedM_p \leftarrow ReceivedM_p \cup (q, s, MsgNum, m, w)$
6:      if $\#t > \frac{2\cdot|w.active|}{3}$ $s.t.(t, s, MsgNum, m, w) \in ReceivedM_p$ then
7:          for all $t \in w.active \setminus s$ do
8:              send (READY, $p, s, MsgNum, m, w$) to $t$
9:          end for
10:     end if
11: end upon
12: upon receiving (READY, $q, s, MsgNum, m, w$) from node $q$
13:     $ReadyM_p \leftarrow ReadyM_p \cup (q, s, MsgNum, m, w)$
14:     if $\#t > \frac{|w.active|}{3}$ $s.t.(t, s, MsgNum, m, w) \in ReadyM_p$ then
15:         for all $t \in w.active \setminus s$ do
16:             send (READY, $p, s, MsgNum, m, w$) to $t$
17:         end for
18:     end if
19:     if $\#t > \frac{2\cdot|w.active|}{3}$ $s.t.(t, s, MsgNum, m, w) \in ReadyM_p$ then
20:         $ValidatedM_p \leftarrow ValidatedM_p \cup (s, MsgNum, m, w)$
21:     end if
22:     $Traverse_p(\emptyset, \perp)$
23:     $ReceiveMsg_p()$
24: end upon
```

As we discussed in previous sections, our protocol is able to be adjusted into any broadcast abstractions. Here we show how to build the reliable broadcast based on our protocols.

Compare to the consistent broadcast, the basic broadcast algorithm is changed from the authenticated echo broadcast algorithm to the authenticated double-echo broadcast algorithm [39]. The algorithm is called authenticated double-echo broadcast because it has two *ECHO* steps. In the first *ECHO* step, node p sends message m with *ECHO* tag to other nodes when it accepts the message m with *SEND* tag from node s. In the second step, if node p receives message m with *ECHO* tag, it stores it in $ReceivedM_p$ (line 5). If more than $\frac{2\cdot|w.active|}{3}$ copies of the message m with *ECHO* tag from other nodes in configuration w are stored in $ReceivedM_p$, node p sends message m with *READY* tag to other nodes in configuration w (line 6-11).

We introduce the second echo procedure in the broadcast algorithm. Nodes broadcast messages with *READY* tag to response the first *echo*. If node p receives a message with *READY* tag, it stores it in $ReadyM_p$ (line 14). If more than $\frac{|w.active|}{3}$ copies of the message m with *READY* tag from other nodes in configuration w are stored in $ReceivedM_p$, node p sends message m with *READY* tag to other nodes in configuration w (line 15-20).

If more than $\frac{2\cdot|w.active|}{3}$ copies of the message m with *READY* tag from other nodes in configuration w are stored in $ReceivedM_p$, node p validates message m in configuration w and stores it in $ValidatedM_p$ (line 21-24), for further delivery.

With the same argument as in Section 6.4, we can prove that the algorithm satisfies the validity, no duplication, and integrity properties of the dynamic Byzantine reliable broadcast.

Generally speaking, if a Byzantine broadcast algorithm works in a static configuration of nodes, it can be directly adjusted to a dynamic Byzantine broadcast algorithm within our protocols.

Lemma 7: Our algorithm satisfies the agreement property.

Proof: The totality property [33] of the authenticated echo broadcast algorithm directly implies that if correct node q delivers a message m with label l from sender p in configuration w, then every correct node eventually delivers a message m with label l from sender p in configuration w.

Correct nodes are stimulated to discover the updated configuration when it receives a message in such a configuration. Our assumption of the number of Byzantine nodes and the correctness of reconfiguration operations ensure that every correct node will eventually visit all established configurations in the system. When a configuration w is visited by node p, it will deliver all messages it validates in w. The agreement property holds intuitively. □

Theorem 2: Algorithm 3 and Algorithm 4 realize a dynamic Byzantine reliable broadcast.

Proof: Because Algorithm 3 satisfies the validity, no duplication, integrity, and consistency properties, and Algorithm 4 satisfies the agreement property. With the modification in Algorithm 4, our protocol realizes a dynamic Byzantine reliable broadcast. □

VII. CONCLUSION

In this paper, we studied the dynamic broadcast problem in asynchronous systems with Byzantine faults. Previous reconfiguration results in asynchronous systems restricted to crash only. They assumed nodes behave correctly during the execution. However, in real distributed systems, some malicious nodes would like to attack the system. It is essential to ensure that the robustness of system is still guaranteed even some nodes are Byzantine.

We discussed dynamic Byzantine algorithms without consensus. We first specified the liveness condition of the dynamic reconfiguration problem with Byzantine failures. We also introduced a dynamic Byzantine consistent broadcast and then discuss how to adjust this protocol to other dynamic broadcast protocols. We showed a dynamic Byzantine reliable broadcast as an example, which could inspire future dynamic Byzantine broadcast primitives in fully asynchronous message-passing systems.

REFERENCES

[1] D. Dolev, I. Keidar, and E. Y. Lotem, "Dynamic voting for consistent primary components," in *Proc. 16th Annu. ACM Symp. Princ. Distrib. Comput.*, 1997, pp. 63–71.

[2] D. Davcev and W. A. Burkhard, "Consistency and recovery control for replicated files," in *Proc. 10th ACM Symp. Operating Syst. Princ.*, 1985, pp. 87–96.

[3] A. El Abbadi and S. N. Dani, "A dynamic accessibility protocol for replicated databases," *Data Knowl. Eng.*, vol. 6, no. 4, pp. 319–332, Jul. 1991.

[4] J.-F. Paris and D. D. E. Long, "Efficient dynamic voting algorithms," in *Proc. 4th Int. Conf. Data Eng.*, 1988, pp. 268–275.

[5] S. Gilbert, "Rambo II: Rapidly reconfigurable atomic memory for dynamic networks," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2003.

[6] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. Schultz, "Automatic reconfiguration for large-scale reliable storage systems," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 2, pp. 145–158, Mar. 2010.

[7] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman, "Reconfigurable distributed storage for dynamic networks," *J. Parallel Distrib. Comput.*, vol. 69, no. 1, pp. 100–116, Jan. 2009.

[8] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "Rambo: A robust, reconfigurable atomic memory service for dynamic networks," *Distrib. Comput.*, vol. 23, no. 4, pp. 225–272, Dec. 2010.

[9] J.-P. Martin and L. Alvisi, "A framework for dynamic Byzantine storage," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2004, pp. 325–334.

[10] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, no. 2, pp. 1–32, Apr. 2011.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[12] A. Shraer, J.-P. Martin, D. Malkhi, and I. Keidar, "Data-centric reconfiguration with network-attached disks," in *Proc. 4th Int. Workshop Large Scale Distrib. Syst. Middleware*, 2010, pp. 22–26.

[13] E. Gafni and D. Malkhi, "Elastic configuration maintenance via a parsimonious speculating snapshot solution," in *Proc. Int. Symp. Distrib. Comput.* Cham, Switzerland: Springer, 2015, pp. 140–153.

[14] L. Jehl, R. Vitenberg, and H. Meling, "SmartMerge: A new approach to reconfiguration for atomic storage," in *Proc. Int. Symp. Distrib. Comput.* Cham, Switzerland: Springer, 2015, pp. 154–169.

[15] J. Sliwinski and R. Wattenhofer, "ABC: Proof-of-stake without consensus," 2019, *arXiv:1909.10926*.

[16] S. Benz and F. Pedone, "Elastic Paxos: A dynamic atomic multicast protocol," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 2157–2164.

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

[18] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable Paxos," Microsoft Res., Redmond, WA, USA, Tech. Rep., 2008. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2008/04/stoppableV9.pdf

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

[20] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *Proc. EuroSys Conf.*, 2006, pp. 103–115.

[21] R. Rodrigues and B. Liskov, "Rosebud: A scalable Byzantine-fault-tolerant storage architecture," Massachusetts Inst. Technol., Cambridge, MA, USA, MIT-LCS-TR-932, 2003.

[22] R. Rodrigues and B. Liskov, "Brief announcement: Reconfigurable Byzantine-fault-tolerant atomic memory," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, 2004, p. 386.

[23] L. Lamport, "The part-time parliament," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 277–317.

[24] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[25] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proc. 7th Int. Conf. Architectural Program. Lang. Operating Syst.*, 1996, pp. 84–92.

[26] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *Proc. OSDI*, vol. 4, 2004, p. 8.

[27] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. OSDI*, 2004, vol. 4, nos. 91–104, pp. 1–14.

[28] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, 2004, pp. 338–346.

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

[30] P. Unterbrunner, G. Alonso, and D. Kossmann, "E-Cast: Elastic multicast," ETH Zurich, Zurich, Switzerland, Tech. Rep. 719, 2011.

[31] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y.-A. Pignolet, D.-A. Seredinschi, and A. Tonkikh, "Dynamic Byzantine reliable broadcast [technical report]," 2020, *arXiv:2001.06271*.

[32] C. Cachin and J. A. Poritz, "Secure INtrusion-tolerant replication on the internet," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 167–176.

[33] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Cham, Switzerland: Springer, 2011.

[34] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine storage," in *Proc. Int. Symp. Distrib. Comput.* Cham, Switzerland: Springer, 2002, pp. 311–325.

[35] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *J. ACM*, vol. 40, no. 4, pp. 873–890, Sep. 1993.

[36] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distrib. Comput.*, vol. 2, no. 2, pp. 80–94, Jun. 1987.

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

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[39] G. Bracha, "Asynchronous byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, Nov. 1987.

JING LI received the Ph.D. degree in economics from the Jiangxi University of Finance and Economics, China, in 2015. She is currently an Associate Professor at the Jiangxi University of Finance and Economics. She has published more than ten journals in the related statistical study fields. Her research interests include data mining and statistical study.

TIANMING YU received the B.S. degree in civil engineering from Nanjing Tech University, in 2021. He is currently pursuing the master's degree in applied statistics with the Jiangxi University of Finance and Economics. His research interests include information systems and financial statistics.

YE WANG received the B.S. degree in microelectronics from Peking University, and the M.S. degree in robotics and the Doctorate of Science degree from ETH Zürich. He is currently an Assistant Professor at the University of Macau. His research interests include blockchain, financial technology, human–computer interaction, and security.

ROGER WATTENHOFER received the Ph.D. degree in computer science from ETH Zürich, Switzerland. He also worked multiple years at Microsoft Research, Redmond, WA, USA; at Brown University, Providence, RI, USA; and at Macquarie University, Sydney, Australia. He is currently a Full Professor at the Information Technology and Electrical Engineering Department, ETH Zürich. He has published in different communities, such as distributed computing (e.g., PODC, SPAA, and DISC), networking and systems (e.g., SIGCOMM, SenSys, IPSN, OSDI, and MobiCom), algorithmic theory (e.g., STOC, FOCS, SODA, and ICALP), and more recently also machine learning (e.g., ICML, NeurIPS, ICLR, ACL, and AAAI). He has published the book *Blockchain Science: Distributed Ledger Technology*, which has been translated to Chinese, Korean, and Vietnamese. His research interests include a variety of algorithmic and systems aspects in computer science and information technology, such as distributed systems, positioning systems, wireless networks, mobile systems, social networks, financial networks, and deep neural networks. His work received multiple awards, such as the Prize for Innovation in Distributed Computing for his work in distributed approximation.