CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 2012-01

Department of Computer Science and Engineering

Chalmers University of Technology

Goeteborg University

Goeteborg, Sweden, 2012

Self-Stabilizing Byzantine Resilient Topology Discovery and Message Delivery

(Technical Report)

Shlomi Dolev * Omri Liba * Elad M. Schiller †

Abstract

Traditional Byzantine resilient algorithms use $2f+1$ vertex disjoint paths to ensure message delivery in the presence of up to f Byzantine nodes. The question of how these paths are identified is related to the fundamental problem of topology discovery.

Distributed algorithms for topology discovery cope with a never ending task, dealing with frequent changes in the network topology and unpredictable transient faults. Therefore, algorithms for topology discovery should be self-stabilizing to ensure convergence of the topology information following any such unpredictable sequence of events. We present the first such algorithm that can cope with Byzantine nodes. Starting in an arbitrary global state, and in the presence of f Byzantine nodes, each node is eventually aware of all the other non-Byzantine nodes and their connecting communication links.

Using the topology information, nodes can, for example, route messages across the network and deliver messages from one end user to another. We present the first deterministic, cryptographic-assumptions-free, self-stabilizing, Byzantine-resilient algorithms for network topology discovery and end-to-end message delivery. We also consider the task of r-neighborhood discovery for the case in which r and the degree of nodes are bounded by constants. The use of r-neighborhood discovery facilitates polynomial time, communication and space solutions for the above tasks.

The obtained algorithms can be used to authenticate parties, in particular during the establishment of private secrets, thus forming public key schemes that are resistant to man-in-the-middle attacks of the compromised Byzantine nodes. A polynomial and efficient end-to-end algorithm that is based on the established private secrets can be employed in between periodical re-establishments of the secrets.

1 Introduction

Self-stabilizing Byzantine resilient topology discovery is a fundamental distributed task that enables communication among parties in the network even if some of the components are compromised by an adversary. Such topology discovery is becoming extremely important nowadays where countries main infrastructures, such as the electrical smart-grid, water supply networks and intelligent transportation systems are subject

*Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel. Email: {dolev, liba}@cs.bgu.ac.il. Partially supported by Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant number 428/11) and Cabarnit Cyber Security MAGNET Consortium.

†Department of Computer Science and Engineering, Chalmers University of Technology, Goeteborg, Sweden. Email: elad@chalmers.se. Partially supported by the EC, through project FP7-STREP-288195, KARYON (Kernel-based ARchitecture for safetY-critical cONtrol) and the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 257007.

to cyber-attacks. Self-stabilizing Byzantine resilient algorithms naturally cope with mobile attacks [e.g., 16]. Whenever the set of compromised components is fixed (or dynamic, but small) during a period that suffice for convergence of the algorithm the system starts demonstrating useful behavior following the convergence. For example, consider the case in which nodes of the smart-grid are constantly compromised by an adversary while local recovery techniques, such as local node reset and/or refresh, ensure the recovery of a compromised node after a bounded time. Once the current compromised set does not imply a partition of the communication graph, the distributed control of the smart grid automatically recovers. Self-stabilizing Byzantine resilient algorithms for topology discovery and message delivery are important for systems that have to cope with unanticipated transient violations of the assumptions that the algorithms are based upon, such as unanticipated violation of the upper number of compromised nodes and unanticipated transmission interferences that is beyond the error correction code capabilities.

The dynamic and difficult-to-predict nature of electrical smart-grid and intelligent transportation systems give rise to many fault-tolerance issues and require efficient solutions. Such networks are subject to transient faults due to hardware/software temporal malfunctions or short-lived violations of the assumed settings for the location and state of their nodes. Fault-tolerant systems that are *self-stabilizing* [5] can recover after the occurrence of transient faults, which can drive the system to an arbitrary system state. The system designers consider *all* configurations as possible configurations from which the system is started. The self-stabilization design criteria liberate the system designer from dealing with specific fault scenarios, risking neglecting some scenarios, and having to address each fault scenario separately.

We also consider Byzantine faults that address the possibility of a node to be compromised by an adversary and/or to run a corrupted program, rather than merely assuming that they start in an arbitrary local state. Byzantine components may behave arbitrarily (selfishly, or even maliciously) as message senders and/or as relaying nodes. For example, Byzantine nodes may block messages, selective omit messages, redirect the route of messages, playback messages, or modify messages. Any system behavior is possible, when all (or one third or more of) the nodes are Byzantine nodes. Thus, the number of Byzantine nodes, f, is usually restricted to be less than one third of the nodes [5, 13].

The task of *r-neighborhood network discovery* allows each node to know the set of nodes that are at most r hops away from it in the communication network. Moreover, the task provides information about the communication links attached to these nodes. The task *topology discovery* considers knowledge regarding the node's entire connected component. The r-neighborhood network discovery and network topology discovery tasks are identical when r is the diameter of the communication graph.

This work presents the first deterministic self-stabilizing algorithms for r-neighborhood discovery in the presence of Byzantine nodes. We assume that every r-neighborhood cannot be partitioned by the Byzantine nodes. In particular, we assume the existence of at least $2f + 1$ vertex disjoint paths in the r-neighborhood, between any two non-Byzantine nodes, where at most f Byzantine nodes are present in the r-neighborhood, rather than in the entire network. [1] Note that by the self-stabilizing nature of our algorithms, recovery is guaranteed after a temporal violation of the above assumption. When r is defined to be the diameter of the communication graph, our assumptions are equivalent to the standard assumption for Byzantine agreement in general (rather than only complete) communication graphs. In particular the standard assumption is that $2f + 1$ vertex disjoint paths exist and *are known* (see e.g., [13]) while we present distributed algorithms to find these paths starting in an arbitrary state.

Related work. Self-stabilizing algorithms for finding vertex disjoint paths for at most two paths between

[1] Section 4 considers cases in which r and the node degree, Δ, are constants. For these case, we have $\mathcal{O}(n)$ disjoint r-neighborhoods. Each of these (disjoint) r-neighborhoods may have up to f Byzantine nodes, and yet the above assumptions, about at least $2f + 1$ vertex disjoint paths in the r-neighborhood, hold.

any pair of nodes, and for all vertex disjoint paths in anonymous mesh networks appear in [1] and in [11], respectively. We propose self-stabilizing Byzantine resilient procedures for finding $f + 1$ vertex disjoint paths in $2f + 1$-connected graphs. In [9], the authors study the problem of spanning tree construction in the presence of Byzantine nodes. Nesterenko and Tixeuil [15] presented a deterministic *non-stabilizing* algorithm for topology discovery in the presence of Byzantine nodes. The authors do not consider the automatic recovery implied by the self-stabilization property. [[Awerbuch and Sipser [3] consider algorithms that were designed for synchronous static network and give topology update as an example. They show how to use such algorithms in asynchronous dynamic networks. Unfortunately, their scheme starts from a consistent state and cannot cope with transient faults or Byzantine.]]

Byzantine gossip [2, 4, 6, 10, 12, 14] and *Byzantine Broadcast* [8, 17] consider the dissemination of information in the presence of Byzantine nodes rather than self-stabilizing topology discovery. Non-self-stabilizing Byzantine resilient gossip in the presence of one selfish node is considered in [2, 12]. In [6], the authors study oblivious deterministic gossip algorithms for multi-channel radio networks with a malicious adversary. They assume that the adversary can disrupt one channel per round, preventing communication on that channel. In [4], the authors consider probabilistic gossip mechanisms for reducing the redundant transmissions of flooding algorithms. They present several protocols that exploit local connectivity to adaptively correct propagation failures and protect against Byzantine attacks. Probabilistic gossip mechanisms in the context of recommendations and social networks are considered in [10]. In [14] the authors consider rules for avoiding a combinatorial explosion in (non-self-stabilizing) gossip protocol. Note that deterministic and self-stabilizing solutions are not presented in [2, 4, 6, 10, 12, 14].

Drabkin et al. [8] consider non-self-stabilizing broadcast protocols that overcome Byzantine failures by using digital signatures, message signature gossiping, and failure detectors. Our deterministic self-stabilizing algorithm merely use the topological properties of the communication graph to ensure that messages dropped or modified by Byzantine nodes will be detected, and retransmitted in a way that guarantees correct delivery to the application layer. A non-self-stabilizing broadcasting algorithm is considered in [17]. The authors assume the restricted case in which links and nodes of a communication network are subject to Byzantine failures, and that faults are distributed randomly and independently.

Our contribution. We present two cryptographic-assumptions-free yet secure algorithms that are deterministic, self-stabilizing and Byzantine resilient.

We start by showing the existence of deterministic, self-stabilizing, Byzantine resilient algorithms for network topology discovery and end-to-end message delivery. [[The algorithms convergence time is in $\mathcal{O}(n)$. They take in to account every possible path and requiring bounded (yet exponential) memory and bounded (yet exponential) communication costs.]] Therefore, we also consider the task of r-neighborhood discovery, where r is a constant. We assume that if the r-neighborhood of a node has f Byzantine nodes, there are $2f + 1$ vertex independent paths between the node and any non-Byzantine node in its r-neighborhood. The obtained r-neighborhood discovery requires polynomial memory and communication costs and supports deterministic, self-stabilizing, Byzantine resilient algorithm for end-to-end message delivery across the network. [[Unlike topology update, the proposed end-to-end message delivery algorithm establishes message exchange synchronization between end-users that is based on message reception acknowledgments.]]

Document structure. Settings and requirements appear in Section 2. The self-stabilizing Byzantine resilient distributed algorithm for topology discovery is presented in Section 3. The end-to-end communication algorithm appears in Section 4. Extensions and concluding remarks appear in Section 5. Detailed proofs appear in the Appendix and in [7].

2 Preliminaries

We consider settings of a standard asynchronous system [cf. 5]. The system consists of a set, $N = \{p_i\}$ of communicating entities, chosen from a set P, which we call *nodes*. The upper bound on the number of nodes in the system is $n = |P|$. Each node has a unique identifier. Sometime we refer to a set, $P \setminus N$, of nonexisting nodes that a false indication on their existence can be recorded in the system. A node p_i can directly communicate with its *neighbors*, $N_i \subseteq N$. The system can be represented by a network of directly communicating nodes, $G = (N, E)$, named the *communication graph*, where $E = \{(p_i, p_j) \in N \times N : p_j \in N_i\}$. We denote N_k's set of indices by $indices(N_k) = \{m : p_m \in N_k\}$ and N_k's set of edges by $edges(N_j) = \{p_j\} \times N_j$.

The r-neighborhood of a node $p_i \in N$ is the connected component that includes p_i and all nodes that can be reached from p_i by a path of length r or less. The r-neighborhood version of the algorithm for network topology discovery considers communication graphs in which the number of neighbors of a node p_i is bounded by a constant Δ. Hence, when both the neighborhood radius, r, and the node degree Δ are constants the number of nodes in the r-neighborhood is also bounded by a constant, namely by [[$\mathcal{O}(\Delta^{r+1})$.]]

We model the communication channel, $queue_{i,j}$, from node p_i to node $p_j \in N_i$ as a FIFO queuing list of the messages that p_i has sent to p_j and p_j is about to receive. When p_i sends message m, the operation send inserts a copy of m to every $queue_{i,j}$, such that $p_j \in N_i$. We assume that the number of messages in transit, i.e., stored in $queue_{i,j}$, is at most $capacity$. Once m arrives, p_j executes receive and m is dequeued.

We assume that p_i is completely aware of N_i, as in [15]. In particular, we assume that the identity of the sending node is known to the receiving one. In the context of the studied problem, we say that node $p_i \in N$ is *correct* if it reports on its genuine neighborhood, N_i. A *Byzantine* node, $p_b \in N$, is a node that can send arbitrarily corrupted messages. Byzantine nodes can introduce new messages and modify or omit messages that pass through them. This way they can, e.g., disinform correct nodes about their neighborhoods, or about the neighborhood of other correct nodes, or the path through which messages travel, to name a very few specific misleading actions that Byzantine nodes may exhibit. We denote by C and B the set of correct, and respectively, Byzantine nodes. We assume that $|B| = f$, the identity of the nodes in B is unknown to the nodes in C. Nevertheless, B is fixed throughout the considered execution segment. These execution segments are long enough for convergence and then for obtaining sufficient useful work. We assume that between any pair of correct nodes there are at least $2f + 1$ vertex disjoints paths. We denote by $G_c = (C, E \cap C \times C)$ the *correct graph* induced by the set of correct nodes.

Self-stabilizing algorithms never terminate (see [5]). The non-termination property can be easily identified in the code of a self-stabilizing algorithm: the code is usually a do forever loop that contains communication operations with the neighbors. An iteration is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters branches).

Every node, p_i, executes a program that is a sequence of *(atomic) steps*. For ease of description, we assume the interleaving model where steps are executed atomically, a single step at any given time. An input event can either be the receipt of a message or a periodic timer going off triggering p_i to send. Note that the system is totally asynchronous and the (non-fixed) node processing rates are irrelevant to the correctness proof.

The *state* s_i of a node p_i consists of the value of all the variables of the node (including the set of all incoming communication channels, $\{queue_{j,i} | p_j \in N_i\}$). The execution of a step in the algorithm can change the state of a node. The term *(system) configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n)$, where each s_i is the state of node p_i (including messages in transit for p_i). We define an *execution* $E = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system configurations $c[x]$ and steps $a[x]$, such that each

configuration $c[x+1]$ (except the initial configuration $c[0]$) is obtained from the preceding configuration $c[x]$ by the execution of the step $a[x]$. We often associate the notation of a step with its executing node p_i using a subscript, e.g., a_i. An execution R (run) is *fair* if every correct node, $p_i \in C$, executes a step infinitely often in R. Time (e.g. needed for convergence) is measured by the number of *asynchronous rounds*, where the first asynchronous round is the minimal prefix of the execution in which every node takes at least one step. The second asynchronous round is the first asynchronous round in the suffix of the run that follows the first asynchronous round, and so on. The message complexity (e.g. needed for convergence) is the number of messages measured in the specific case of synchronous execution.

We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold. A configuration c is a *safe configuration* for an algorithm and the task of LE provided that any execution that starts in c is a legal execution (belongs to LE). An algorithm is *self-stabilizing* with relation to the task LE when every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

3 Topology Discovery

The topology discovery is based on accumulating messages from vertex disjoint paths. Each message contains an ordered list of nodes it passed so far, starting in a source node, and a neighborhood, which is the set of nodes, claimed to be directly connected to the source.

Each node p_i periodically sends a message to each neighbor. The message sent contains the local topology, a source i and an empty path. The arrival of a message m to p_i triggers an insert of m to $informedTopology_i$ and a consistency test of the content of $informedTopology_i$. The consistency test results in storing local topologies for which there are enough independent evidence in a result array. The result array is initialized just prior to the consistency test. The consistency test of p_i iterates over each node p_k such that, p_k appears in at least one of the messages stored in $informedTopology_i$. For each such node p_k, node p_i checks whether there are at least $f+1$ messages from the same source node that have mutually vertex disjoint paths and report on the same neighborhood. The neighborhood of each such p_k, that has at least $f+1$ vertex disjoint paths with identical neighborhood, is accumulated in $Result[k]$. Moreover, the total number of paths [[that]] relayed this neighborhood is kept in $Count[k]$.

We note that there may still be nodes $p_{fake} \in P \setminus (C \cup B)$, for which there is an entry $Result[fake]$. For example, $informedTopology$ may contain f messages, all originated from different Byzantine nodes, and a message m' that appears in the initial configuration and supports the (false) neighborhood the Byzantine messages refer to. These $f+1$ messages can contain mutually vertex disjoint paths, and thus during the consistency test, a result will be found for $Result[fake]$. We show that during the next computations, the message m' will be identified and ignored.

The $Result$ set should include two reports for each (undirected) edge; the two nodes that are attached to the edge, each send a report. Hence, $Result$ includes a set of directed (report) edges. The term *contradicting edge* is needed when examining the $Result$ set consistency.

Definition 1 (Contradicting edges) *Given two nodes, $p_i, p_j \in P$, we say that the edge (p_i, p_j) is* contradicting with the set $Neighborhood_j \subseteq edges(N_j)$, *if* $(p_i, p_j) \notin Neighborhood_j$.

Following the consistency test, p_i examines the $Result$ array for contradictions. Node p_i checks the path of each message $m \in informedTopology_i$ with source p_r, neighborhood $neighborhood_r$ and $Path_r$. If every edge (p_s, p_j) on the path appears in $Result[s]$ and $Result[j]$, then we move to the next message. Otherwise, we found a fake supporter, and therefore we reduce $Count[r]$ by one. In case the resulting $Count[r]$ is smaller than $f+1$, we nullify the r'th entry of the $Result$ array. Once all messages were

processed, the $Result$ array consisting of the (confirmed) local topologies is the output. At the end p_i forwards the arriving message m to each neighbor that does not appear in the path of m. The message sent by p_i includes the node from which m arrived as part of the path m.

The pseudocode appears in Algorithm 1. In every iteration of the infinite loop, p_i starts to compute its preliminary topology view by calling $ComputeResults$ in line 2. Then, every node p_k in the queue $InformedTopology$, node p_i goes over the messages in the queue from head to bottom. While iterating the queue, for every message m with source p_k, neighborhood N_k and visited path $Path_k$, p_i inserts $Path_k$ to $opinion[N_k]$, see line 18. After inserting, p_i checks if there is a neighborhood $Neig_k$ for which $opinion[Neig_k]$ contains at least $[[f+1]]$ vertex disjoint paths, see line 19. When such a neighborhood is found, it is stored in the $Result$ array (line 19). In line 20, p_i stores the number of vertex disjoint paths relayed messages that contained the selected neighborhood for p_k. After computing an initial view of the topology, in line 3, p_i removes non-existing nodes from the computed topology. For every message m in $InformedTopology$, node p_i aims at validating its visited path. In line 24, p_i checks if there exists a node p_k whose neighborhood contradicts the visited path of m. If such a node exists, p_i decreases the associated entry in the $Count$ array (line 25). This decrease may cause $Count[r]$ to be smaller than $f+1$, in this case p_i considers p_k to be fake and deletes the local topology of p_k from $Result[r]$ (line 26).

Upon receiving a message m, node p_i inserts the message to the queue, in case it does not already exist, and just moves it to the top of the queue in case it does. The node p_i now needs to relay the message p_i got to all neighbors that are not on the message visited path (line 9). When sending, p_i also attaches the identifier of the node, from which the message was received, to the visited path of the message.

Algorithm's correctness proof. We now prove that within a linear amount of asynchronous rounds, the system stabilizes and every output is legal. The proof considers an arbitrary starting configuration with arbitrary messages in transit that could be actually in the communication channel or already stored in p_j's message queue and will be forwarded in the next steps of p_j. Each message in transit that traverse correct nodes can be forwarded within less than $\mathcal{O}(|C|)$ asynchronous rounds. Note that any message that traverses Byzantine nodes and arrives to a correct node that has at least one Byzantine node in its paths. The reason is that the correct neighbor to the last Byzantine in the path lists the Byzantine node when forwarding the message. Thus, f is at most the number of messages that encode vertex disjoint paths from a certain source that are initiated or corrupted by a Byzantine node. Since there are at least $f+1$ vertex disjoint paths with no Byzantine nodes from any source p_k to any node p_i and since p_k repeatedly sends messages to all nodes on all possible paths, p_i receives at least $f+1$ messages from p_k with vertex disjoint paths.

The usage of the FIFO queue and the repeated send operations of p_k ensure that the most recent $f+1$ messages with vertex disjoint paths in $InformedTopology$ queue are uncorrupted messages. Namely, misleading messages that were present in the initial configuration will be pushed to appear below the new $f+1$ uncorrupted messages. Thus, each node p_i eventually has the local topology of each correct node (stored in the $Result_i$ array). The opposite is however not correct as local topologies of non-existing nodes may still appear in the result array. For example, $InformedTopology_i$ may include in the first configuration $f+1$ messages with vertex disjoint paths for a non-existing node.

Since after $ComputeResults$ we know the correct neighborhood of each correct node p_k, we may try to ensure the validity of all messages. For every message that encodes a non-existing source node, there must be a node p_ℓ on the message path, such that p_ℓ is correct and p_ℓ's neighbor is non-existing, this is true since p_i itself is correct. Thus, we may identify these messages and ignore them. Furthermore, no valid messages are ignored because of this validity check.

We also note that, since we assume that the nodes of the system are a subset of P. The size of the queue $InformedTopology$ is bounded. Next, we bound the amount of memory of a node. The details of the correctness and convergence proofs appear in the Appendix and in [7].

Algorithm 1: Topology discovery, code for node p_i

Input: $Neighborhood_i$: The ids of the nodes with which node p_i can communicate directly;
Output: $ConfirmedTopology \subset P \times P$: Discovered topology, which is represent by a directed edge set;
Variable $InformedTopology : Queue$, see Figure 1: topological messages, $\langle node, neighborhood, path\rangle$;
Function: $NodeDisjointPaths(S)$: Test $S = \{\langle node, neighborhood, path\rangle\}$ to encode at least $f + 1$ vertex disjoint paths;
Function: $PathContradictsNeighborhood(k, Neighborhood_k, path)$: Test that there is no node $p_j \in N$ for which there is an edge (p_k, p_j) in the message's visited path, $path \subseteq P \times N$, such that (p_k, p_j) is contradicting with $Neighborhood_k$;

1 **while true do**
2 $\quad Result \leftarrow ComputeResults()$
3 $\quad$ **let** $Result \leftarrow RemoveContradictions(Result)$
4 $\quad RemoveGarbage(Result)$
5 $\quad ConfirmedTopology \leftarrow ConfirmedTopology \cup (\bigcup_{p_k \in P} Result[k])$
6 $\quad$ **foreach** $p_k \in N_i$ **do send**$(i, Neighborhood_i, \emptyset)$ **to** p_k

7 **Upon Receive** $(\langle \ell, Neighborhood_\ell, VisitedPath_\ell\rangle)$ **from** p_j;
begin
8 $\quad Insert(p_\ell, Neighborhood_\ell, VisitedPath_\ell \cup \{j\})$
9 $\quad$ **foreach** $p_k \in N_i$ **do if** $k \notin VisitedPath_\ell$ **then send**$(p_\ell, Neighborhood_\ell, VisitedPath_\ell \cup \{j\})$ **to** p_k

10 **Procedure:** $Insert(k, Neighborhood_k, VisitedPath_k)$;
begin
11 $\quad$ **if** $\exists m = \langle \ell, Neighborhood_\ell, VisitedPath_\ell\rangle \in InformedTopology : (\ell, Neighborhood_\ell, VisitedPath_\ell) = (k, Neighborhood_k, VisitedPath_k)$ **then** $InformedTopology.MoveToHead(m)$
12 $\quad$ **else if** $p_k \in N \wedge Neighborhood_k \subseteq indices(N) \wedge VisitedPath_k \subseteq indices(N)$ **then** $InformedTopology.Insert(\langle k, Neighborhood_k, VisitedPath_k\rangle)$

13 **Function:** $ComputeResults()$;
begin
14 $\quad$ **foreach** $p_k \in P : \langle k, Neighborhood_k, VisitedPath_k\rangle \in InformedTopology$ **do**
15 $\quad\quad$ **let** $(FirstDisjointPathsFound, Message, opinion[]) \leftarrow (false, InformedTopology.Iterator(), [\emptyset])$
16 $\quad\quad$ **while** $Message.hasNext()$ **do**
17 $\quad\quad\quad \langle \ell, Neighborhood_\ell, VisitedPath_\ell\rangle \leftarrow Message.Next()$
18 $\quad\quad\quad$ **if** $\ell = k$ **then** $opinion[Neighborhood_\ell].Insert(\langle \ell, Neighborhood_\ell, VisitedPath_\ell\rangle)$
19 $\quad\quad\quad$ **if** $FirstDisjointPathsFound = false \wedge NodeDisjointPaths(opinion[Neighborhood_\ell])$ **then**
$\quad\quad\quad\quad (Result[k], FirstDisjointPathsFound) \leftarrow (Neighborhood_\ell, \mathbf{true})$
20 $\quad\quad Count[k] \leftarrow opinion[Result[k.SizeOf()]]$
21 $\quad$ **return** $Result$

22 **Function:** $RemoveContradictions(Result)$;
begin
23 $\quad$ **foreach** $\langle r, Neighborhood_r, VisitedPath_r\rangle \in InformedTopology$ **do**
24 $\quad\quad$ **if** $\exists p_k \in P : PathContradictsNeighborhood(p_k, Result[k], VisitedPath_r) = \mathbf{true}$ **then**
25 $\quad\quad\quad$ **if** $Neighborhood_r = Result[r]$ **then** $Count[r] \leftarrow Count[r] - 1$
26 $\quad\quad\quad$ **if** $Count[r] \leq f$ **then** $Result[r] \leftarrow \emptyset$
27 $\quad$ **return** $Result$

28 **Procedure:** $RemoveGarbage(Result)$;
begin
29 $\quad$ **foreach** $p_k \in N$ **do**
30 $\quad\quad$ **foreach** $m = \langle k, Neighborhood_k, VisitedPath_k\rangle \in InformedTopology :$
$\quad\quad \{k\} \cup Neighborhood_k \cup VisitedPath_k \not\subseteq P \vee InformedTopology.IsAfter(m, opinion[k][Result[k]])$ **do**
$\quad\quad InformedTopology.Remove(m)$

Lemma 1 (Bounded memory) *Let $p_i \in C$ be a correct node. At any time, there are at most $n \cdot 2^{2n}$ messages in $InformedTopologyany_i$, where $n = |P|$ and $\mathcal{O}(|P| \log(|P|))$ is the message size.*

r-neighborhood discovery. Algorithm 1 demonstrates the existence of a deterministic self-stabilizing Byzantine resilient algorithm for topology discovery. Lemma 1 shows that the memory costs are high when the entire system topology is to be discovered. We note that one may consider the task of r-neighborhood

- $Insert(m)$: Insert item m to the head of the queue.
- $Remove(Messagem)$: Remove item m from the queue.
- $Iterator()$: Returns an pointer for iterating over the queue's elements by the order in which they reside in the queue.
- $HasNext()$: Tests whether the Iterator is at the end of the queue.
- $Next()$ Returns the next element to iterate over.
- $SizeOf()$ Returns the number of elements in the calling set.
- $MoveToHead(m)$: Move item m to the head of the queue.
- $IsAfter(m, S)$: Test that item m is after the items $m' \in S$, where S is a set of items in the queue.

Figure 1: $Queue$: general purpose data structure for queuing items, and its operation list.

discovery. Recall that in the r-neighborhood discovery task, it is assumed that every r-neighborhood cannot be partitioned by Byzantine nodes. Therefore, it is sufficient to constrain the maximal path length in line 9. The correctness proof of the algorithm for the r-neighborhood discovery follows similar arguments to the correctness proof of Algorithm 1.

4 End-to-End Delivery

We use the discovered network topology to design a self-stabilizing Byzantine resilient algorithm for the transport layer protocol. Namely, using the repeatedly collected topology information for implementing end-to-end communication between (not necessarily neighboring) nodes. In this context, we face the challenge of finding $f + 1$ correct vertex disjoint paths and the need to propose efficient solutions for different system settings.

The value of $ConfirmedTopology$ is a set of directed edges (p_i, p_j). An undirected edge is approved if both (p_i, p_j) and (p_j, p_i) appear in $ConfirmedTopology$. An edge is said to be suspected, whenever only one edge (in one direction) appears in $ConfirmedTopology$. The sender has to choose $2f + 1$ vertex independent paths to the receiver. If there exists at least one such set of paths then the sender can safely use them to communicate with the receiver (similar to Algorithm 1). However, the collected topology may not include even one such set of $2f + 1$ vertex independent paths. The reason is that f of the paths that should appear in the collected topology may be controlled by Byzantine nodes. Namely, the information about at least one edge in each such path may not arrive to the sender.

We propose three procedures for overcoming this difficulty in different system setting. The first procedure assumes f is a constant. Thus, the sender may apply the following procedure for selecting a set of vertex disjoint paths $Paths$, that contains $f + 1$ correct paths. For each possible choice of f nodes $p_1, p_2, \ldots p_f$ in the system, the sender computes a new graph G' which is the result of removing $p_1, p_2, \ldots p_f$, from G_{out}, the graph defined by the collected topology. The sender now computes a set $\mathcal{P}$ of vertex disjoint paths, where $|\mathcal{P}| = f + 1$, if such a set exists. For each such set $\mathcal{P}$, the sender relays the current message on all paths in $\mathcal{P}$. First we show that this procedure only sends message through a polynomial number of paths. There are $\mathcal{O}(n^f)$ possibilities for choosing f nodes from the system. Thus, $\mathcal{O}(n^f)$ sets of paths are computed, and since f is a constant, this number is polynomial. Moreover, each such set contains at most $f + 1$ paths, because p_i only computes a set $\mathcal{P}$ of size $f + 1$. Thus, in total, the sender sends the message on at most a polynomial number of paths. We now show that this procedure ensures that the message is sent on a sufficient amount of correct paths i.e., $f + 1$. Consider the permutation in which the set of f chosen nodes actually contains the set of Byzantine nodes in the system. Thus G' contains only correct nodes. Furthermore, at least $f+1$ paths that were present in G_{out} are still present in G', since we removed f nodes. Hence, in G', there are at least $f+1$ correct vertex disjoint paths. As stated, the sender chooses a set of paths of size $f + 1$. Each of these paths is correct, and therefore the sender sends the message on at least $f + 1$ correct

vertex disjoint paths as needed.

The second procedure assumes that r and Δ are both constants. The sender sends the message over all possible paths to the receiver. This is feasible only when considering r-neighborhoods, rather than the entire connected component, where the neighborhood radius, r, and the node degree Δ are constants. Next, we present a polynomial solution for the case in which f, r and Δ are not constants, assuming that Byzantine nodes are not directly connected.

The third procedure assumes that Byzantine nodes cannot be immediate neighbors and that all neighbors of a given Byzantine node refer to the Byzantine with the same identifier. Our polynomial cost solution considers the (extended) graph, G_{ext}, that includes all the edges in $confirmedTopology$ and *suspicious edges*, see Definition 2.

Definition 2 (Suspicious edges) *Given three nodes, $p_i, p_j, p_k \in P$, we say that node p_i considers the undirected edge (p_k, p_j) suspicious, if the edge appears as a directed edge in $ConfirmedTopology_i$ for only one direction, e.g., (p_j, p_k).*

The extended graph, G_{ext}, may contain fake edges that Byzantine nodes reports on their existence. Nevertheless, G_{ext} includes all the correct paths of the communication graph, G. Therefore, the $2f + 1$ vertex disjoint paths that exists in G also exists in G_{ext}. These $2f + 1$ paths facilitate our polynomial cost solution.

The sender uses the chosen paths to repeatedly forward the message m that should arrive to the receiver. The sender uses a label to identify the messages. Roughly speaking, the receiver deliver a message received at least $c \cdot n + 1$ consecutive times from $f + 1$ vertex independent paths (according to the path carried in the message). Once the receiver delivers a message to the network layer, the receiver starts to repeatedly send acknowledgments with the label of the delivered message over $2f + 1$ vertex disjoint paths. In addition, the receiver also restarts its counters and the log of received messages upon a message delivery to the network layer. Similarly the sender count acknowledgments to the current label used, when the sender receives at least $c \cdot n + 1$ acknowledgments on a set of $f + 1$ vertex disjoint paths, the sender fetches the next message from the network layer, changes the label and starts to send the new message. We note that starting from an arbitrary configuration, the sender eventually fetches a message from the network layer. This is obvious since if the sender is sending the same message forever, then the receiver counters on $f + 1$ paths must exceed $c \cdot n + 1$. From this point the receiver sends acknowledgments with the correct label forever ensuring that the sender fetches the next message.

The pseudocode of the algorithm appears in Algorithm 2. In every iteration of the infinite loop, p_i fetches a message (line 3). Following the fetch, p_i prepares the label for the next message (line 4). Once the label is ready, p_i starts sending the message over $2f + 1$ vertex disjoint messages which p_i calculates in the procedure $ByzantineFaultToleranceSend(Message)$. When p_i gets enough acknowledgments regarding the current message (see line 5), p_i stops sending the current message and fetches another message.

Upon receiving a message m, node p_i checks in line 7 whether p_i is the destination of the message. If not, p_i forwards the message to the next node on the intended path of the message, not forgetting to update the visited path. If however p_i is the destination of the message, p_i checks the type of the message in line 10. If the type of the message is *Data* then (in line 11) p_i inserts the message payload and label to the part of the data structure associated with the message source, i.e., the sender, and the message visited path. In line 27, node p_i checks whether $2f + 1$ vertex disjoint paths relayed the message at least $capacity \cdot n + 1$ times, where $capacity$ is an upper bound on the number of messages in transit over a communication link. If so, p_i delivers the message to the above layer (line 20), clears the entire data structure and finally sends acknowledgments on $2f + 1$ vertex disjoint paths until a new message is confirmed. Moreover, in line 21 we signal that we are ready to receive a new message. If the type of the message is ACK, we act almost as when the message is of type $Data$. When the condition in line 18 holds, we signal that the message was

confirmed at the receiver by setting $Approved$ to be $true$, in line 18.

Correctness proof. Let us consider three labels, 0, 1, and 2 that are used by the sender in a round robin fashion. Whenever at least $c \cdot n + 1$ identical messages arrive at the receiver with the same label on each of $f + 1$ vertex independent paths, the receiver delivers them, nullify the counters, empty queues and send acknowledges with the label of the delivered message over $2f + 1$ vertex-disjoint paths (cf. line 13). The sender clears counters and queues whenever the sender changes label.

First we prove that the sender fetches infinitely often, by assuming it is not and proving that eventually the receiver sends acknowledgments with the label used by the sender. Hence, the sender must fetch (see Lemma 13). Then in between the second and the fourth fetch of any four successive fetches, where without the loss of generality, the first fetch is with label 0, the second with 1, the third with label 2 and the fourth with 0 the receiver clears its counter and the last fetched message in this sequence that is with label 0 is later delivered.

Following the fetch of each of the first three messages and before the next one, the sender must count $c \cdot n + 1$ acknowledgments with the current label that the sender uses to send, namely with 0, 1 and 2. Since the sender reset the counters when changing the sending label to 1, the receiver must send at least one acknowledgment with label 1 and then with label 2, following the corresponding fetches. Thus, the receiver must clear its counters at least once following the second fetch and before the fourth fetch and then start sending acknowledgments with label 2. After clearing the counters by the receiver and starting sending acknowledgments with label 2 a message with label 0 that is next to be sent, must be delivered and no other message can be counted as arriving at least $c \cdot n + 1$ times through $f + 1$ vertex-disjoint paths. Detailed proof appears in the Appendix and in [7].

Note that the code of Algorithm 2 considers only one possible pair of source and destination. A many-source to many-destination version of this algorithm can simply use a separate instantiation of this algorithm for each pair of source and destination.

5 Extensions and Conclusions

As extension, we suggest to combine the algorithms for r-neighborhood network discovery and the end-to-end capabilities in order to allow the use of end-to-end message delivery within the r-neighborhoods. These two algorithms can be used by the nodes, under reasonable node density assumptions, for discovering their r-neighborhoods and then extending the scope of their end-to-end capabilities beyond their r-neighborhood, as we sketch next. We instruct further remote nodes to relay topology information, and in this way collect information on remote neighborhoods. One can consider an algorithm for studying specific remote neighborhood that are defined, for example, by their geographic region, assuming the usage of GPS inputs; a specific direction and distance from the topology exploring node defines the exploration goal. The algorithm nominates $2f + 1$ nodes in the specific direction to return further information towards the desired direction. The sender uses end-to-end communication to the current $2f + 1$ nodes in the *front* of the current exploration, asking them for their r-neighborhood, chooses a new set of $2f + 1$ nodes for forming a new front. It then instructs each of the current nodes in the current front to communicate with each node in the chosen new front, to nominate the new front nodes to form the exploration front.

To ensure stabilization, this interactive process of remote information collection should never stop. Whenever the current collection process investigates beyond the closest r-neighborhood, we concurrently start a new collection process in a pipeline fashion. The output is the result of the last finalized collection process. Thus, having a correct output after the first time a complete topology investigation is finalized.

In this work we presented two deterministic, self-stabilizing Byzantine-resilience algorithms for topology discovery and end-to-end message delivery. We have also considered an algorithm for discovering r-neighborhood in polynomial time, communication and space. Lastly, we mentioned a possible extension for exploring and communicating with remote r-neighborhoods using polynomial resources as well.

The obtained end-to-end capabilities can be used for communicating the public keys of parties and establish private keys, in spite of f corrupted nodes that may try to conduct man-in-the-middle attacks, an attack that the classical Public key infrastructure (PKI) does not cope with. Once private keys are established encrypted messages can be forwarded over any specific $f + 1$ node independent paths, one of which must be Byzantine free. The Byzantine free path will forward the encrypted message to the receiver while all corrupted messages will be discarded. Since our system should be self-stabilizing, the common private secret should be re-established periodically.

References

[1] F. M. Al-Azemi and M. H. Karaata. Brief announcement: A stabilizing algorithm for finding two edge-disjoint paths in arbitrary graphs. In X. Défago, F. Petit, and V. Villain, editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 433–434. Springer, 2011.

[2] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. C. Li, R. van Renesse, and G. Trédan. How robust are gossip-based communication protocols? *Operating Systems Review*, 41(5):14–18, 2007.

[3] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks (preliminary version). In *FOCS*, pages 206–220. IEEE Computer Society, 1988.

[4] M. Burmester, T. V. Le, and A. Yasinsac. Adaptive gossip protocols: Managing security and redundancy in dense ad hoc networks. *Ad Hoc Networks*, 5(3):313–323, 2007.

[5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[6] S. Dolev, S. Gilbert, R. Guerraoui, and C. C. Newport. Gossiping in a multi-channel radio network. In A. Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[7] S. Dolev, O. Liba, and E. M. Schiller. Self-stabilizing byzantine resilient topology discovery and message delivery. Technical Report 2012:01, Chalmers University of Technology, 2012. ISSN 1652-926X.

[8] V. Drabkin, R. Friedman, and M. Segal. Efficient byzantine broadcast in wireless ad-hoc networks. In *DSN*, pages 160–169. IEEE Computer Society, 2005.

[9] S. Dubois, T. Masuzawa, and S. Tixeuil. Maximum metric spanning tree made byzantine tolerant. In D. Peleg, editor, *DISC*, volume 6950 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2011.

[10] Y. Fernandess and D. Malkhi. On spreading recommendations via social gossip. In F. Meyer auf der Heide and N. Shavit, editors, *SPAA*, pages 91–97. ACM, 2008.

[11] R. Hadid and M. H. Karaata. An adaptive stabilizing algorithm for finding all disjoint paths in anonymous mesh networks. *Computer Communications*, 32(5):858–866, 2009.

[12] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar gossip. In *OSDI*, pages 191–204. USENIX Association, 2006.

[13] N. Lynch. *Distributed Computing*. Morgan Kaufmann Publishers, 1996.

[14] Y. Minsky and F. B. Schneider. Tolerating malicious gossip. *Distributed Computing*, 16(1):49–68, 2003.

[15] M. Nesterenko and S. Tixeuil. Discovering network topology in the presence of byzantine faults. *IEEE Trans. Parallel Distrib. Syst.*, 20(12):1777–1789, 2009.

[16] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *PODC*, pages 51–59. ACM, 1991.

[17] M. Paquette and A. Pelc. Fast broadcasting with byzantine faults. *Int. J. Found. Comput. Sci.*, 17(6):1423–1440, 2006.

Algorithm 2: Self-stabilizing Byzantine resilient end-to-end delivery, code for node p_i.

Interface: $FetchMessage()$: Get a new message from the upper layer. We denote by $InputMessageQueue$ the unbounded queue of all messages that are to be delivered to the destination;
Interface: $DeliverMessage(Source, Message)$: Deliver an arriving message to the higher layer. We denote by $OutputMessageQueue$ the unbounded queue of all messages that are to be delivered to the higher layer. We assume that it always contains at least the last message inserted to it;
Input: $ConfirmedTopology$: The discovered topology, which is represent by a set of directed edges included in $P \times P$, see Algorithm 1;
Data Structure: Transport layer messages: $\langle Source, Destination, VisitedPath, IntentedPath, ARQLabel, Type, Payload\rangle$, where $Source$ is the sending node, $Destination$ is the target node, $VisitedPath$ is the actual relay path, $IntentedPath$ is the planned relay path, $ARQLabel$ is the sequence number of the stop-and-wait ARQ protocol, and $Type \in \{Data, ACK\}$ message type, where DATA and ACK are constant;
Field: $Payload$: the message data;
Variable $Message$: the current message being sent;
Variable $ReceivedMessages[j][Path]$: queue of p_j's messages that were relayed over path $Path$ (see Figure 1);
Variable $Confirmations[j][Path]$: queue of p_j's message acknowledgments that were relayed over path $Path$ (see Figure 1);
Variable $label$: the current sequence number of the stop-and-wait ARQ protocol;
Variable $Approved$: A Boolean variable indicating whether $Message$ was accepted at the destination;
Function: $NodeDisjointPaths(S)$: Test S, a set of paths, to encode at least $f + 1$ vertex disjoint paths;
Function: $FloodedPath(MessageQueue, m)$: Test whether m is encoded by the first $capacity \cdot n + 1$ messages in $MessageQueue$, where $capacity$ is an upper bound on the number of messages in transit over a communication link.;
Function: $SuspiciousEdges()$: Get the set of suspicious edges;
Function: $getDisjointPaths(Topology, Source, Destination)$: Get a set of $f + 1$ vertex disjoint paths between source and destination in the graph induced by Topology.;
Function: $ClearQueue(Source)$: Delete all data in $ReceivedMessages[Source][*]$;
Function: $ClearAckQueue(Destination)$: Delete all data in $Confirmations[Destination][*]$;

1 **while true do**
2 $\quad ClearAckQueue(Message.Destination)$
3 $\quad Message \leftarrow FetchMessage()$
4 $\quad label \leftarrow label + 1 \; modulo \; 3$
5 $\quad$ **while** $Approved =$ **false do** $ByzantineFaultToleranceSend(Message)$

6 **Upon Receive** (msg) **From** p_j;
begin
7 $\quad$ **if** $msg.Destination \neq i$ **then**
8 $\quad\quad msg.VisitedPath \leftarrow msg.VisitedPath \cup \{j\}$
9 $\quad\quad$ **send**(msg)
10 $\quad$ **else if** $msg.Type = Data$ **then**
11 $\quad\quad ReceivedMessages[msg.Source][msg.VisitedPath].insert(\langle msg.Payload, msg.ARQLabel\rangle)$
12 $\quad\quad$ **if** $\exists m \in ReceivedMessages[msg.Source][*] : Paths = \{Path : FloodedPath(ReceivedMessages[msg.Source][Path], m)\} \wedge NodeDisjointPaths(Paths) \wedge msg.source = m.source$ **then**
13 $\quad\quad\quad Confirm(msg.Source, m.ARQLabel, m.Payload)$
14 $\quad\quad\quad NewMesssage =$ **true**
15 $\quad$ **else if** $msg.Type = ACK$ **then**
16 $\quad\quad$ **if** $label = msg.ARQLabel$ **then**
$\quad\quad\quad Confirmations[msg.Source][msg.VisitedPath].insert(\langle msg.Payload, msg.ARQLabel\rangle)$
17 $\quad\quad$ **let** $Paths \leftarrow \{Path : FloodedPath(Confirmations[msg.Source][Path], \langle msg.Payload, msg.ARQLabel\rangle)\}$
18 $\quad\quad$ **if** $NodeDisjointPaths(Paths)$ **then** $Approved =$ **true**

19 **Function:** $Confirm(Source, ARQLabel, Payload)$;
begin
20 $\quad$ **if** $CurrentLabel \neq ARQLabel$ **then** $DeliverMessage(Source, Payload)$
21 $\quad (CurrLbl, NewMessage) \leftarrow (ARQLbl,$ **false**$)$
22 $\quad ClearQueue(Source)$
23 $\quad$ **while** $NewMessage =$ **false do** $ByzantineFaultToleranceSend(\langle Source, ARQLabel, ACK, Payload\rangle)$

24 **Function:** $ByzantineFaultToleranceSend(Destination, ARQLabel, Type, Payload)$;
begin
25 $\quad$ **let** $Paths \leftarrow getDisjointPaths(ConfirmedTopology \cup SuspiciousEdges(), i, Destination)$
26 $\quad$ **foreach** $Path \in Paths$ **do** **send**$(\langle i, Destination, \emptyset, Path, ARQLabel, Type, Payload\rangle)$ **to first**$(Path)$

A Correctness of Algorithm 1

Lemma 1 (Bounded memory) *Let $p_i \in C$ be a correct node. At any time, there are at most $n \cdot 2^{2n}$ messages in $InformedTopologyany_i$, where $n = |P|$ and $\mathcal{O}(|P|\log(|P|))$ is the message size.*

Proof. The queue $InformedTopologyany_i$, is made up of messages in the form $\langle node, neighborhood, visitedpath\rangle$. All nodes that appear in the message, i.e., in the first, second or third entry of the tuple are in N. The first entry, i.e. the node name is one of n possibilities. The second and third entries are subsets of N. Thus each of them has 2^n possibilities. In total there can be at most $2^n \cdot 2^n \cdot n$ messages in every $InformedTopologyany_i$. ■

Definition 3 specifies the requirements of the network topology discovery task. Definition 4 considers correct paths and Definition 5 considers uncorrupted graph topology messages.

Definition 3 (Legal output) *Given correct node $p_i \in C$, we say that p_i's output is* legal*, if it encodes graph $G_{output} = (V_{out}, E_{out})$: (1) $C \subseteq V_{out} \subseteq C \cup B \subseteq N$, and (2) $(E \cap (C \times C)) \subseteq E_{out} \subseteq (E \cap (C \times C)) \cup (B \times (C \cup B)) \subseteq N \times N$.*

Definition 4 (Correct path) *We say $path \subseteq N$ is a* correct *one if all its nodes are correct, i.e., $path \subseteq C$.*

Definition 5 (Valid message) *In Algorithm 1, we refer to a message $m = \langle k, Neighborhood_k, VisitedPath_k\rangle$ as a* valid *message when: (1) $p_k \in C$ and $VisitedPath_k$ encodes a correct path in the communication graph, G, that starts in p_k, and (2) $Neighborhood_k = indices(N_k)$.*

Lemma 2 shows that eventually correct paths do not relay non valid messages. Namely, invalid messages can only exist as the result of: (1) Byzantine interventions that corrupt messages, or (2) transient faults, which occur only prior to the arbitrary starting configuration considered. [2]

Lemma 2 (Eventually valid messages) *Let R be a fair execution of Algorithm 1 that starts in an arbitrary configuration. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the system reaches a configuration after which only valid messages are relayed on correct paths.*

Proof. Let $c \in R$ be the starting configuration. Suppose that c includes an invalid message, $m = \langle \ell, Neighborhood_\ell, VisitedPath_\ell\rangle$, in transit between correct nodes. The lemma is obviously correct for the case that m is relayed by Byzantine nodes during the first $\mathcal{O}(|C \cup B|)$ asynchronous rounds of R. Therefore, we consider only the correct paths, $path$, over which m is relayed during the first $\mathcal{O}(|C \cup B|)$ asynchronous rounds of R. We show that, within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, no correct node in $path$ relays m.

Let $p_j, p_i \in path$ be correct neighbors on the correct path. Suppose that in c, message m is in transit from p_j to p_i. Upon the arrival of message m to p_i (line 7), p_i sends $m_i = \langle \ell, Neighborhood_\ell, VisitedPath_\ell \cup \{j\}\rangle$ to any neighbor $p_k \in path$ on the path for which $p_k \in N_i \wedge k \notin VisitedPath_\ell$, see line 9.

Node p_i adds p_j's identifier to m's visited path $VisitedPath_\ell$, see line 9. The same argument holds for any correct neighbors, $p'_j, p'_j \in path$ when p_j sends message m'_j to the next node in $path$, node p'_i.

[2] This is a common way to argue about self-stabilization, we consider executions that start in an arbitrary configuration that follows the last transient fault, recalling that if additional transient faults occur a new arbitrary configuration is reached from which automatic convergence starts.

Therefore, within $|path \setminus VisitedPath_\ell|$ asynchronous rounds, it holds that $N'_i \cap (path \setminus VisitedPath_\ell) = \{p'_j, p'_i\}$.

Note that p'_i makes sure that $VisitedPath'_\ell$ does not encode loops, i.e., $p_k \notin VisitedPath'_\ell$, see line 9. Therefore, node p'_i does not relay message m' to p_k. ■

Definition 6 considers queues that their recent valid messages encode at least $f+1$ vertex disjoint paths. Moreover, the invalid ones encode at most f such paths.

Definition 6 (Valid queue) *Let $p_i, p_k \in C$ be two correct nodes. We say that p_i's queue, $InformedTopology_i$, is* valid (with respect to p_k) *whenever there is a prefix, $ValidInformation_{i,k}$, of messages m_k in the queue $InformedTopology_i$, such that: (1) there is a subset, $Valid = \{m_\ell = \langle k, Neighborhood_k, VisitedPath_\ell\rangle : m_\ell \text{ is valid}\} \subseteq ValidInformation_{i,k}$, for which the set $\{VisitedPath_\ell\}$ encodes at least $f+1$ vertex disjoint paths, and (2) the set, $Invalid = \{m_\ell = \langle k, Neighborhood_k, VisitedPath_\ell\rangle : m_\ell \text{ is invalid}\} \subseteq ValidInformation_{i,k}$, for which the set $\{VisitedPath_\ell\}$ encodes at most f vertex disjoint paths.*

Claim 3 shows that, within $\mathcal{O}(|C|)$ asynchronous rounds, correct paths propagate valid messages.

Claim 3 *Let $path \subseteq C$ be a correct path from p_i to p_k. Suppose that $m_i = \langle i, N_i, \emptyset\rangle$ is a (valid) message that p_i sends, see line 6. Within $\mathcal{O}(|path|)$ asynchronous rounds, message m_i is relayed on $path$, and arrives at p_k as $m'_i = \langle i, N_i, path\rangle$. Namely, $path$ is m'_i's visited path.*

Proof. Let $c \in R$ be the first configuration that follows the start of m_i's propagation in $path$. I.e., c is the configuration that immediately follows the step in which node p_i sends m_i by executing line 6. Let $p_r, p_j \in path$ be two correct neighbors on the path. Without the loss of generality, suppose that node p_i sends message m_i directly to node p_r, i.e., in c, node p_r is just about to receive m_i. The proof arguments hold also when assuming that p_j sends message $m_j = \langle i, N_i, \{r\}\rangle$ to the next node in $path$. Thus, generality is not lost.

We show that, within one asynchronous round, p_r sends m_r to p_j. Upon the arrival of message m_i to p_r (line 7), node p_r sends the message m_r to any neighbor, such as p_j, for which $p_j \in N_r \wedge r \notin VisitedPath_i = \emptyset$, see line 9. Since the same argument holds when p_j sends m_j to the next node in $path$, we have that within $|path|$ asynchronous rounds, m'_i is delivered to node p_k.

□

Lemma 4 shows that queues get to become valid.

Lemma 4 (Eventually valid queues) *Let R be a fair execution of Algorithm 1 that starts in an arbitrary configuration and $p_i, p_k \in C$ be any pair of correct nodes. The system reaches a configuration in which the queue, $InformedTopology_i$, is valid (with respect to p_k), within $\mathcal{O}(|C \cup B|)$ asynchronous rounds.*

Proof. Let $c \in R$ be a configuration achieved in Lemma 2 within $\mathcal{O}(|C \cup B|)$ asynchronous rounds. We show that within $\mathcal{O}(|C \cup B|)$ asynchronous rounds after c, the system reaches a configuration in which $InformedTopology_i$, is *valid (with respect to p_k)*, see Definition 6.

In configuration c, all messages in transit on correct paths are valid, see Lemma 2. Thus, the only messages entering $InformedTopology_i$ are either valid or have passed through Byzantine nodes. Denote $m_{barrier}$ to be the top message the queue $InformedTopology_i$. Moreover, $ValidInformation_{i,k}$ includes all the messages in $InformedTopology_i$, that are between the queue's head and $m_{barrier}$.

We show that condition (1) of Definition 6 holds. There are $2f+1$ vertex disjoint paths between p_i and p_k. At most f nodes are Byzantine and thus, there are at least $f+1$ vertex disjoint paths between p_i and p_k that are correct. By Claim 3 within $\mathcal{O}(|C|)$ asynchronous rounds, a valid message, m_k, is received on all

$f+1$ (correct) vertex disjoint paths. Message m_k is inserted to $InformedTopology_i$ after configuration c. Therefore, m_k is in front of $m_{barrier}$. Hence, the set $Valid = \{m_\ell = \langle \ell, Neighborhood_\ell, VisitedPath_\ell \rangle : m_\ell \text{ is valid}\} \subseteq ValidInformation_{i,k}$ contains at least $f+1$ valid messages whose respective visited paths, $VisitedPath_\ell$, are vertex disjoint.

We show that condition (2) of Definition 6 holds. Any invalid messages, m_k, that is sent after configuration c, must go through a Byzantine node, see Lemma 2.

Claim 5 *Suppose that message m is relayed through a Byzantine node after configuration c, then in any following configuration, while m is still in transit, there is a Byzantine node in the visited path.*

Proof. Observe the first correct node p_k after the last Byzantine node b on m's path. p_k is correct, thus it inserts b to the visited path. b is the last on the path and so the visited path must contain it until end of transit or passing through a different Byzantine. □

Each such Byzantine node is recorded in the message path, see Claim 5. Since there are at most f Byzantine nodes, there could be at most f such messages with vertex disjoint paths. This completes the proof condition (2) and the lemma. ■

Lemma 7 shows that correct information gets confirmed, and requires Definition 7.

Definition 7 (Message confirmation) *We say that message $m_i = \langle k, Neighborhood_k, VisitedPath_{k_i} \rangle$ is* confirmed (by node p_i) *when $Neighborhood_k \subseteq ConfirmedTopology_i$.*

Lemma 6 (Eventually confirmed messages) *Let R be a fair execution of Algorithm 1 that starts in an arbitrary configuration and $p_i, p_k \in C$ be any pair of correct nodes. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the system reaches a configuration after which the fact that message $m_i = \langle k, Neighborhood_k, VisitedPath_{k_i} \rangle$ is confirmed, implies that $Neighborhood_k = indices(N_\ell)$.*

Proof. Let $c \in R$ be the first configuration in which $InformedTopology_i$ is a valid queue and node p_i completes a full iteration of the do forever loop that starts in line 1. By Lemma 4, the system reaches c within $\mathcal{O}(|C \cup B|)$ asynchronous rounds.

We how that in configuration c, the array $Result_i$ satisfies that $Result_i[k] = indices(N_\ell)$. We go through the computation of $Result$ in lines 2 to 4.

- $ComputeResults()$**,** line 2. Let $Res_i[k] = indices(N'_\ell)$ be $ComputeResults()$'s return value with respect to node p_k. We show that $Res_i[k] = indices(N_\ell)$. Moreover, we show that the neighborhood that will be found will be that which is represented in $Valid = \{m_\ell = \langle k, Neighborhood_k, VisitedPath_\ell \rangle : m_\ell \text{ is valid}\} \subseteq ValidInformation_{i,k}$.

We recall that the set $\{VisitedPath_\ell\}$ encodes at least $f+1$ disjoint paths. Also in the prefix $ValidInformation_{i,k}$ one can not find $f+1$ invalid messages with vertex disjoint messages; See Definition 6.

The function must choose the message containing the neighborhood $Neighborhood_k$. Otherwise, we have chosen a different neighborhood for k, say $Neighborhood'_k \neq Neighborhood_k = indices(N_k)$. That is, at the time of checking line 19 with neighborhood $Neighborhood_\ell = Neighborhood'_k$, there were at least $f+1$ vertex disjoint paths in $opinion[Neighborhood_\ell]$. This is in contradiction to condition (2) of Definition 6. Moreover in line 20, it holds $Count[k] > f+1$, since at least all the correct paths were counted.

- $RemoveContradictions()$, line 3. Let $Res_i = ComputeResults()$ and $ResRemoveContradictions_i = RemoveContradictions(Res_i)$ (line 3). We show that $ResRemoveContradictions_i[r] = indices(N_r)$. The function $RemoveContradictions()$ modifies

$Res_i[r]$ only in line 26 by nullifying it whenever $Count[r] \leq f$. We demonstrate that, for any correct path $VisitedPath_k$, there exists no p_ℓ for which $PathContradictsNeighborhood(p_\ell, Res_i[\ell], VisitedPath_k) = \mathbf{true}$, which is the condition in line 24.

We explain that there is no node p_ℓ and a contradicting edge (p_j, p_ℓ) with the set $Res_i[\ell]$. By the assumption that $VisitedPath_k$ is correct and that node $p_\ell \in VisitedPath_k$, we have that $p_\ell \in C$ is correct. Thus $Res_i[\ell] = indices(N_\ell)$, see previous item of this claim on $ComputeResults()$. $VisitedPath_k$ is correct, and therefore (p_j, p_ℓ) must be in $VisitedPath_k$.

- $RemoveGarbage()$, line 4. This procedure does not modify $Res_i = RemoveContradictions(ComputeResults())$. We have shown that $Result_i[k] = indices(N_k)$. Thus, only the correct neighborhood is confirmed for every correct node p_k. ■

Lemma 7 shows that eventually there are no fake nodes.

Lemma 7 (Eventually no fake nodes) *Let R be a fair execution of Algorithm 1 that starts in an arbitrary configuration, $p_j \in N$ be any node, and $p_\ell \in P \setminus (C \cup B)$ be a node that is not included in the communication graph, G. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the system reaches a configuration after which $(p_j, p_\ell) \notin ConfirmedTopology_i$*

Proof. Let $c \in R$ be the configuration reached within $\mathcal{O}(|C \cup B|)$ asynchronous rounds according to Lemma 6. For any correct node, $p_i \in C$, we show that in c, the execution of $RemoveContradictions()$ results in $Count_i[\ell] \leq f$ and nullifies $Result_i[\ell]$.

We start by showing that for every path p that relays a message which encodes the set $Result_i[\ell]$, and does not contain Byzantine nodes, a contradiction is found in $RemoveContradictions()$. Namely, the if conditions of line 24 holds.

Note that, p may not be a correct path even though it contains no Byzantine nodes. For example p may contain nodes p_z that are not even in the communication graph, i.e., $p_z \in P \setminus (C \cup B)$.

Let $p_r \in C \cup B$ be the first correct node on path p. Such a node exists, because p_i is correct and on the path p. Since p_r is correct, after the execution of $ComputeResults()$, we have that p_r's neighborhood, N_r, is encoded in $Result_i[r]$, see Lemma 6.

Denote the last edge in the path (p_r, p_s), where $p_s \in P \setminus (C \cup B)$. Note that node p_s is not a node in the system and since $Result_i[r]$ encodes N_r's neighborhood, we have that $p_s \notin Result_i[r]$. Thus, the edge (p_r, p_s) is contradicting with the set $Result_i[r]$. Namely, by the condition in line 24, we have that line 25 must decrease $Count[\ell]$.

We note that immediately before the function $RemoveContradictions()$ returns, the integer $Count[\ell]$ may count only incorrect paths, which contain at least one Byzantine node. Since there are at most f Byzantine nodes, $Count[\ell] \leq f$ as needed. ■

Theorem 8 demonstrates the self-stabilization properties.

Theorem 8 (Self-stabilization) *Let R be a fair execution of Algorithm 1 that starts in an arbitrary configuration and $p_i \in C$ be a correct node. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the system reaches a safe configuration after which p_i's output is always legal, see Definition 3.*

Proof. The systems reaches configuration $c \in R$ of Lemma 6 within $\mathcal{O}(|C \cup B|)$ asynchronous rounds. We show that c is a safe configuration by showing that the output is legal, we must show that $ConfirmedTopology_i$ encodes a graph $G_{output} = (V_{out}, E_{out})$, such that: (1) $C \subseteq V_{out}$, (2) $(E \cap (C \times C)) \subseteq E_{out}$, (3) $V_{out} \subseteq C \cup B \subseteq N$, and (4) $E_{out} \subseteq (E \cap (C \times C)) \cup (B \times (C \cup B)) \subseteq P \times N$.

For every correct node $p_k \in C$, we have that N_k is confirmed in c, see Lemma 6. Thus, $p_k \in V_{out}$ and condition (1) holds.

Let (p_j, p_k) be an edge in the communication graph between two correct nodes, we show $(p_j, p_k) \in E_{out}$. Since p_j is correct, it is inserted to $ConfirmedTopology_i$, see Lemma 6. Thus, $(p_j, p_k) \in edges(N_j) \wedge edges(N_j) \subseteq ConfirmedTopology_i$ in c, thus condition (2) holds as well.

There is no $p_\ell \in P \setminus (C \cup B)$ and node $p_j \in N$, such that $(p_\ell p_j) \in ConfirmedTopology_i$, see Lemma 7. Thus, $V_{out} \subseteq C \cup B \subseteq N$ and $E_{out} \subseteq (E \cap (C \times C)) \cup (B \times (C \cup B)) \subseteq P \times N$. I.e., conditions (3) and (4) hold in c. ■

B Correctness of Algorithm 2

Lemma 9 shows that senders and receivers can eventually find at least $2f + 1$ vertex-disjoint paths between them. Note that at least $f + 1$ of them are correct.

Lemma 9 *Let R be a fair execution of Algorithm 2 that starts in an arbitrary configuration and $p_s, p_r \in C$ a pair of correct nodes (sender and receiver). Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds the system reaches a configuration in which the set $ConfirmedTopology \cup SuspiciousEdges$ encodes a set of $2f + 1$ vertex disjoint paths from p_s to p_r and at least $f + 1$ of them are correct.*

Proof. Let c be a safe configuration with respect to Algorithm 1. Let $Paths = getDisjointPaths(ConfirmedTopology \cup SuspiciousEdges(), i, Destination)$ be a set of disjoint paths in c, as in line 25, where $i \in \{s, r\}$. We first show that $\mid Paths \mid \geq 2f + 1$ before showing that at least $f + 1$ of them are correct.

We consider the graph $G' = (N, E_{G'})$, which is computed from $ConfirmedTopology$ and the suspicious edges in c. We demonstrate that G' contains the real communication graph, G. Let $e = (p_j, p_k) \in E_{G'}$. When p_j and p_k are both correct, $e \in G'$ since c is safe. When p_j is correct and p_k is Byzantine, we must consider the cases in which p_k reports, and does not report, e as part of its local neighborhood. Namely, either $e \in ConfirmedTopology$, or $e \in SuspiciousEdges()$, because p_k does not report about e, but p_i does. Since $G \subseteq G'$, G' must contain $2f + 1$ vertex disjoint paths between any p_s and p_r, because G does. Thus $\mid Paths \mid \geq 2f + 1$.

Moreover, the same arguments implies that there may be at most f incorrect paths, which contain at least one Byzantine node. Hence, there are at least $f + 1$ correct nodes in $Paths$. ■

Definitions 8, 9 and 10 are needed for lemmas 11, 12 and 13.

Definition 8 (Confirmation) *Given configuration c, we say that message m is* confirmed (by the receiver) *when $m \in OutputMessageQueue$.*

Definition 9 (Approve) *Given fair execution, R, of Algorithm 2, we say that message $m = \langle Source, Destination, VisitedPath, IntentedPath, ARQLabel, DATA, Payload \rangle$ is* being approved (by the sender p_{Source}) *during the first atomic step, a_{sender}, in which the sender executes line 18, where $Source = sender$ $ARQLabel = msg_{sender}.ARQLabel$ and $Payload = msg_{sender}.Payload$, see line 17. Denote by $c_{approved}$ the configuration that immediately follows a_{sender}. Given configuration c that appears after $c_{approved}$ in R, we say that message m* is approved (by the sender) *in configuration c.*

Definition 10 (Clear-sender-receiver) *Given configuration c, we say that the sender is* clear (with respect to the receiver)*, if the queue $Confirmations[receiver] = \emptyset$ in c. Moreover, the receiver is* clear (with respect to the sender) *, if the queue $ReceivedMessages[sender] = \emptyset$ in c.*

Claim 10 shows that a message that is relayed on a correct path is received at the destination within $\mathcal{O}(|C \cup B|)$ asynchronous rounds. Moreover, the destination receives the message with correct visiting set.

Claim 10 *Let R be a fair execution of Algorithm 2 that starts in a safe configuration, c, with respect to Algorithm 1. Let $p_{source}, p_{dest} \in C$ be pair of correct nodes. Let c_{send} be the configuration immediately following a step in which p_{source} sends message Msg on a correct path $Path = p_{source}, p_1, p_2, \ldots p_{dest}$ from source, p_{source}, to destination, p_{dest}. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, p_{dest} receives Msg with a visiting set containing all nodes on $Path$ except p_{dest}.*

Proof. Upon the arrival of message m to p_k (line 6), node p_i asserts that he is not the destination, p_{dest}, (line 7). Immediately after, p_i sends the message m to the next neighbor, p_{i+1}, see line 9. Since the same argument holds when p_j sends m to the next node in $path$, we have that within $|Path|$ asynchronous rounds, m is delivered to node p_{dest}. ■

Claim 11 says that when the sender repeatedly sends message Msg, for a duration of at least $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the receiver eventually confirms message Msg.

Claim 11 *Let R be a fair execution of Algorithm 2 that starts in a safe configuration, c, with respect to Algorithm 1. Let $p_s, p_r \in C$ be a pair of correct sending and receiving nodes. Suppose that, for a duration of at least $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, p_s's steps include only the execution of the function $ByzantineFaultToleranceSend(Msg)$ in the loop of line 5. Within that period, the system reaches configuration $c_{receive}$ in which p_r confirms Msg.*

Proof. Denote c_{send} as the configuration immediately following the first step in which p_s sends message Msg in R, see line 27. Within $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, the first frame containing Msg arrives at p_r, see Claim 10. Moreover, after another $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, every correct path relays message Msg at least $\mathcal{O}(capacity \cdot |C \cup B|)$ times. This is correct since every asynchronous round, p_s sends a new frame containing Msg on each of the $2f + 1$ vertex disjoint paths. Moreover, by Claim 10, the last frame sent on all $2f + 1$ paths arrives after another $\mathcal{O}(capacity \cdot |C \cup B|)$.

Assume, in the way of proof by contradiction, that Msg is not confirmed by p_r. This implies that the queues, $ReceivedMessages[p_s][*]$, in p_r containing messages sent from p_s were not cleared at least since c_{send}, see line 22. Thus, p_r contains $capacity \cdot n + 1$ indications of Msg on $f + 1$ vertex disjoint paths. Denote c_{last} as the configuration immediately after the arrival of the $(capacity \cdot n + 1)$-th frame of the $f + 1$'th path to relay $capacity \cdot n + 1$ frames containing Msg. Immediately after c_{last}, p_s must go through line 12, because the conditions in line 12 hold. Thus, a contradiction and Msg is confirmed within $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds. ■

Claim 12 says that when the receiver is sending acknowledgments about a message, that message eventually becomes approved. We note that Claim 12 considers acknowledgments sent from the receiver to the sender, rather than messages sent from the sender to the receiver, as in Claim 11.

Claim 12 *Let R be a fair execution of Algorithm 2 that starts in a safe configuration, c, with respect to Algorithm 1. Let $p_s, p_r \in C$ be a pair of correct sending and receiving nodes. Suppose that, for a duration of at least $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, p_r's steps include only the execution of the function $ByzantineFaultToleranceSend(Ack)$ in the loop of line 23. That is, p_r is sending acknowledgments on message Msg. Within that period, the system reaches configuration $c_{receive}$ in which p_s approves Msg, see Definition 9.*

Proof. Denote c_{send} as the configuration immediately following the first step in which p_r sends acknowledgment Ack in R, see line 23. Within $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, the first frame containing Ack arrives at p_s, see Claim 10. Moreover, after another $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, every correct path relays message Ack at least $\mathcal{O}(capacity \cdot |C \cup B|)$ times. This is correct since every asynchronous round, p_r sends a new frame containing Ack on each of the $2f + 1$ vertex disjoint paths. Moreover, by Claim 10, the last frame sent on all $2f + 1$ paths arrives after another $\mathcal{O}(capacity \cdot |C \cup B|)$.

The queues, $Confirmations[p_r][*]$ are cleared only when a message sent to p_r is approved, see line 2. Since, p_r is acknowledging the current message, Msg, by sending Ack, the only message that can be approved is Msg. This is true since each path, $Path$, may contain at most $capacity \cdot |C \cup B|$ acknowledgments for other messages in the path queues.

Assume, in the way of proof by contradiction, that Msg is not approved by p_s. By the arguments above, p_s's queues, $Confirmations_s[p_r][*]$, which contains p_r's acknowledgments that p_s received, were not cleared at least since c_{send}, see line 2. Thus, p_s contains $capacity \cdot n + 1$ indications of Ack on $f + 1$ vertex disjoint paths. Denote c_{last} as the configuration immediately after the arrival of the $(capacity \cdot n + 1)$-th frame of the $f + 1$'th path to relay $capacity \cdot n + 1$ frames containing Ack. Immediately after c_{last}, p_s must go through line 18, because the conditions in line 18 hold. Thus, a contradiction and Msg is approved within $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds. ■

Lemma 13 shows that the senders repeatedly fetch messages.

Lemma 13 *Let R be a fair execution of Algorithm 2 that starts in a safe configuration, c, with respect to Algorithm 1. Let $p_s, p_r \in C$ be pair of correct sending and receiving nodes. Moreover, c_ℓ is the configuration that immediately follows the ℓ-th time in R in which p_s fetches a message from the input queue. For every ℓ, the system reaches c_ℓ within $\mathcal{O}(\ell \cdot |C \cup B|)$ asynchronous rounds.*

Proof. By the code of Algorithm 2, on every iteration of the do forever loop (lines 2 to 5), a message is fetched in line 3. This do forever loop includes another loop in line 5. We prove the lemma by showing that the loop of line 5 is completed within $\mathcal{O}(|C \cup B|)$ asynchronous rounds.

The proof considers the case in which the sender, p_s, does not wait in line 5 for a long time before considering the case in which p_s does wait. We show that for the latter case, the receiver, p_r, confirms p_s's current message. After confirming the message, the receiver, p_r, begins sending acknowledgments to the sender, p_s. The proof shows that after the acknowledgments are sent, p_s approves the message and fetches a new one. We show this by considering the case in which p_r repeatedly sends acknowledgments for a sufficient amount of time, and a case in which it does not.

Suppose that p_s does not wait in line 5 more than $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds. In this case, p_s starts the infinite loop again within $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, and fetch a new message, see line 3. Thus, for the case in which p_s does not wait in line 5 more than $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, the lemma is correct.

Suppose that p_s is executing line 5 and waits for acknowledgments on message Msg for more than $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds. Thus, p_s floods $2f + 1$ vertex-disjoint paths with the message Msg, see Claim 9. Eventually, the receiver, p_r, receives message Msg for $\mathcal{O}(capacity \cdot |C \cup B|)$ times on $f + 1$ vertex-disjoint paths and confirms Msg, see Claim 11. After confirming it, the receiver sends acknowledgments on $2f + 1$ vertex-disjoint paths until confirming a new message Msg_{new}. This is true because the condition in line 23 holds only when a new message is confirmed, see line 14.

Let us consider the case in which, during $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, message Msg_{new} does not arrive to the receiver. By Claim 12, eventually the sender receives the acknowledgments for

$capacity \cdot n + 1$ times on $f + 1$ vertex disjoint paths. Claim 12 also says that the sender considers the message accepted by the receiver. In line 18, the sender assigns $Approved = \mathbf{true}$. Thus, the condition in line 5 holds and the sender fetches the next message, see line 3. Hence, the system reaches configuration c_{fetch} that immediately follows a step in which the sender, p_s, fetches the next message. Thus, for the case in which, during $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, message Msg_{new} does not arrive to the receiver, the lemma is correct.

We continue by considering the case in which, during $\mathcal{O}(capacity \cdot |C \cup B|)$ asynchronous rounds, message Msg_{new} does arrive to the receiver. Let c_{conf} be the configuration that immediately follows the step in which p_r confirms Msg. Since the receiver confirms Msg, we have that p_r is clear (with respect to the sender) in configuration c_{conf}, see Definition 10 and line 22.

If Msg_{new} was sent by the sender, it must have been fetched after c, and c_{fetch} is reached when message Msg_{new} is fetched. It may be the case however, that Msg_{new} was not sent by the sender. Message Msg_{new} was confirmed by $2f + 1$ vertex disjoint paths. Since there are at most f Byzantines, at least one of these paths, $Path$, must be correct. Moreover, in c_{conf}, the receiver is clear, thus the $capacity \cdot n + 1$ that p_r counts in $ReceivedMessages[p_s][*]$ have all been received after configuration c_{conf}. Note that the sender sends at least one of these messages, because at most $capacity \cdot n$ messages could be in the edges of $Path$ at any given configuration. Thus the sender sends Msg_{new}, which p_s fetches immediately before c_{fetch}. I.e., the system reaches c_{fetch}. ■

Theorem 8 says that, starting from that fourth (or even the third) message that the sender fetches, the receiver confirms the sender's messages. The proof of Theorem 8 is based on Lemma 14, which says that, in every sequence of four messages that the sender is fetching, the receiver confirms the fourth (or even the third) message.

Lemma 14 *Let R be a fair execution of Algorithm 2 that starts in a safe configuration, c_{start}, with respect to Algorithm 1. Let c_h be a configuration that immediately follows the h-th step in which the sender fetches the h-th input queue message, m_h. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the receiver confirms message m_4.*

Proof.

Claim 15 *In c_2, the sender is clear (with respect to the receiver), see Definition 10.*

Proof. By definition, c_2 immediately follows atomic step a_2, in which, after clearing the confirmation queue in line 2, the sender fetches message m_2 and sends it. □

Claim 16 *Between the configurations c_3 and c_4, there is a configuration $c_{receiver-clear}$ in which the receiver is clear (with respect to the sender).*

Proof. Suppose, without the loss of generality, that immediately after $c_{sender-clear}$, the sender is waiting for a message with label 1. By lemma 13, the sender eventually fetches the next message. The sender can only fetch a new message once $Approved$ is true, see line 5. Moreover, $Approved$ is only set to $true$ once the queue $Confirmations[receiver][*]$ contains $2f + 1$ flooded paths, see line 18. Thus, the sender counts $2f + 1$ vertex disjoint paths that relayed acknowledgments with label 1. Moreover, the sender is clear in $c_{sender-clear}$. Hence, configuration $c_{sender-clear}$ contains no message in $Confirmations[receiver][*]$. Starting from $c_{sender-clear}$, the sender receives $capacity \cdot n + 1$ acknowledgments on $2f + 1$ vertex disjoint paths for the current message with label 1. Note that at least one of these $2f + 1$ paths, $Path$, is correct,

because there are f Byzantine. Since $|Path| \leq n$ and each edge on $Path$ may contain at most $capacity$ messages, we have that at least one of the acknowledgments that includes $Path$ as its visiting path, is sent by the receiver between $c_{sender-clear}$ and configuration $c_{receiver-send} \in R$. We show that $c_{receiver-send} = c_{receiver-clear}$.

This means that after $c_{sender-clear}$, the sender clears the confirmations queue, $Confirmations[receiver][*]$, and fetches the next message, assigning it the label 2, see lines 2 through line 5. By similar arguments, we know that the receiver sends at least one acknowledgment with label 2.

To conclude, there is a configuration $c \in R$ in which the receiver is sending acknowledgments with label 1, and then a configuration c' in which the receiver sends acknowledgments with label 2. Moreover, between two consecutive executions of line 23, the receiver has to go through line 22. Thus, the receiver cleared it's message queues, $Confirmations[sender][*]$, immediately before configuration $c_{receiver-clear}$ and $c_{receiver-send} = c_{receiver-clear}$. □

Let us consider configuration $c_{receiver-clear}$ from the end of proof of Claim 16.

The next message to be sent after $c_{receiver-clear}$, is m_4, the message fetched in c_4, with label 0. Between $c_{receiver-clear}$ and c_4, all messages sent by the sender have the label 2. By arguments stated above, the message, m, that is the next message to be confirmed after $c_{receiver-clear}$, must have been sent by the sender at least once since $c_{receiver-clear}$. The sender, sends only messages with label 0 and 2. Moreover, the last message to be confirmed had a label 2. Thus, $CurrentLabel = 2$, see line 21. Any sent message with label 2 is not inserted to the confirmations queue, $Confirmations[sender][*]$ between $c_{receiver-clear}$ and the configuration that immediately follows the next sender's fetch, see line 20. Thus, by line 4, the next message to be confirmed is a message with label 0, which must be m_4. ■

Theorem 8 (Self-stabilization) *Let R be a fair execution of Algorithm 2 that starts in an arbitrary configuration. Within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, the system reaches a safe configuration c after which: (1) the receiver confirms message m in step $a_r^m \in R$, and (2) for every step a_r^m, there is a corresponding step, $a_s^m \in R$, that occurs before a_r^m and in which the sender sends m.*

Proof. Let c be the configuration that Claim 16 denote as c_4, which the system reaches within $\mathcal{O}(|C \cup B|)$ asynchronous rounds, see Lemma 13. Let m_i be the i-th message fetched.

Suppose that $i \geq 4$. Lemma 14 considers the four consecutive messages $m_{i-3}, \ldots m_i$ and says that the receiver confirms message m_i. Thus, condition (1) holds.

Condition (2) follows from arguments similar to the ones used in the proof of Lemma 11. Namely, for the case of $i \geq 5$, message m_{i-1} is confirmed, see lemma 14. Immediately after the receiver confirms m_{i-1}, it clears the queue $ReceivedMessages[sender][*]$, see lines 20 to 22. Thus, there exists a configuration $c_{receiver-clear}$ in which the receiver is clear (with respect to the sender) before c_i, see Definition 10. Moreover, a message is confirmed only if the queue $ReceivedMessages[sender][*]$ contains $2f + 1$ flooded paths, see line 12. These flooded paths implies that in configuration c_i, the queue $ReceivedMessages[sender][*]$ contains $capacity \cdot n + 1$ indications of m_i on $2f + 1$ node disjoint paths. Thus, m_i is confirmed only after a period that follows $c_{receiver-clear}$ and includes its reception at least $capacity \cdot n + 1$ times on each of the $2f + 1$ vertex disjoint paths.

Recall that we assume that there are at most f Byzantine nodes in the system. At least one path, $Path$, of the above $2f + 1$ paths is correct. Moreover, $|Path| \leq n$ and each edge on $Path$ may contain at most $capacity$ messages. Thus, at least one of the $capacity \cdot n + 1$ message that were relayed on the correct path $Path$ was sent by the sender. This completes the correctness proof. ■