Self-Stabilizing End-to-End Communication in

Bounded Capacity, Omitting, Duplicating and non-FIFO Dynamic Networks ⋆

(Extended Abstract)

Shlomi Dolev[1], Ariel Hanemann[1], Elad M. Schiller[2], and Shantanu Sharma[1]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Israel. {dolev, hanemann, sharmas}@cs.bgu.ac.il. ⋆⋆

[2] Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. elad@chalmers.se. ⋆⋆⋆

Abstract. End-to-end communication over the network layer (or data link in overlay networks) is one of the most important communication tasks in every communication network, including legacy communication networks as well as mobile ad hoc networks, peer-to-peer networks and mash networks. We study end-to-end algorithms that exchange packets to deliver (high level) messages in FIFO order without omissions or duplications. We present a self-stabilizing end-to-end algorithm that can be applied to networks of bounded capacity that omit, duplicate and reorder packets. The algorithm is network topology independent, and hence suitable for always changing dynamic networks with any churn rate.

1 Introduction

End-to-end communication is a basic primitive in communication networks. A *sender* must transmit messages to a *receiver* in an exactly once fashion, where no omissions, duplications and reordering are allowed. Errors occur in transmitting packets among the network entities – one significant source of error is noise in the transmission media. Thus, error detection and error correcting techniques are employed as an integral part of the transmission in the communication network. These error detection and correction codes function with high probability. Still,

⋆ **Accepted in** 14^{th} **International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2012).** Also appears as a technical report in [10]

⋆⋆ 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), Cabarnit Cyber Security MAGNET Consortium, Grant from the Institute for Future Defense Technologies Research named for the Medvedi of the Technion, and Israeli Internet Association.

⋆⋆⋆ Work was partially supported by the EC, through project FP7-STREP-288195, KARYON (Kernel-based ARchitecture for safetY-critical cONtrol).

when there is a large volume of communication sessions, the probability that an error will not be detected becomes high, leading to a possible malfunction of the communication algorithm. In fact, it can lead the algorithm to an arbitrary state from which the algorithm may never recover unless it is *self-stabilizing* [8]. By using packets with enough distinct labels infinitely often, we present a self-stabilizing end-to-end communication algorithm that can be applied to dynamic networks of bounded capacity that omit, duplicate and reorder packets.

Contemporary communication and network technologies enhance the need for automatic recovery and interoperability of heterogeneous devices and the means of wired and wireless communications, as well as the churn associated with the totally dynamic communication networks. Having a self-stabilizing, predictable and robust basic end-to-end communication primitive for these dynamic networks facilitates the construction of high-level applications. Such applications are becoming extremely important nowadays where countries' main infrastructures, such as the electrical smart-grid, water supply networks and intelligent transportation, are based on cyber-systems. Defining the communication network as a bounded capacity network that allows omissions, duplications and reordering of packets and building (efficient) exactly once message transmission using packets, allows us to abstract away the exact network topology, dynamicity and churn.

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* [8,7] 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, the risk of neglecting some scenarios, and having to address each fault scenario separately.

Related work and our contribution End-to-end communication and data-link algorithms are fundamental for any network protocol [25]. End-to-end algorithms provide the means for message exchange between senders and receivers over unreliable communication links. Not all end-to-end communication and data-link algorithms assume initial synchronization between senders and receivers. For example, Afek and Brown [1] present a self-stabilizing alternating bit protocol (ABP) for FIFO packet channels without the need for initial synchronization. Self-stabilizing token passing was used as the bases for self-stabilizing ABP over unbounded capacity and FIFO preserving channels in [17,11]. Spinelli [24] introduced two self-stabilizing sliding window ARQ protocols for unbounded FIFO channels. Dolev and Welch [15] consider the bare network to be dynamic networks with FIFO non-duplicating communication links, and use source routing over paths to cope with crashes. In contrast, we

do not consider known network topology nor base our algorithms on a specific routing policy. We merely assume bounded network capacity.

In [2], an algorithm for self-stabilizing unit capacity data link over a FIFO physical link is assumed. Flauzac and Villai [16] describe a snapshot algorithm that uses bidirectional and FIFO communication channels. Cournier et al. [5] consider a snap-stabilizing algorithm [3] for message forwarding over message switched network. They ensure one time delivery of the emitted message to the destination within a finite time using destination based buffer graph and assuming underline FIFO packet delivery.

In the context of dynamic networks and mobile ad hoc networks, Dolev, Schiller and Welch [14,12,13] presented self-stabilizing algorithms for token circulation, group multicast, group membership, resource allocation and estimation of network size. Following [14,12,13], similar approaches to cope with constantly changing networks have been investigated [22] in addition to other fundamental problems such as clock synchronization [21], dissemination [18,20], leader election [19,6,4], and consensus [23] to name a few. In this paper, we investigate the basic networking tasks of end-to-end communication over the network layer (or overlay networks), that are required for the design of fundamental problems, such as the aforementioned problems considered in [21,22,18,20,19,6,4,23].

Recently, Dolev et al. [9] presented a self-stabilizing data link algorithm for reliable FIFO message delivery over bounded non-FIFO and non-duplicating channel. This paper presents the first, to the best of our knowledge, self-stabilizing end-to-end algorithms for reliable FIFO message delivery over bounded non-FIFO and *duplicating* channel.

Due to space limit, some of the proofs are omitted from this extended abstract and can be found in [10].

2 System Settings

We consider a distributed system that includes *nodes* (or processors), $p_1, p_2, \ldots, p_N$. We represent a distributed system by a *communication graph* that may change over time, where each processor is represented as a node. Two *neighboring* processors, p_i and p_j, that can exchange packets directly are connected by a link in the communication graph. Packet exchange between neighbors is carried via (directed) communication links, where packets are sent from p_i to p_j through the directed link (p_i, p_j) and packets are sent from p_j to p_i through (p_j, p_i), the opposite directed link. End-to-end communication among non-neighbor nodes, p_s and p_r, is facilitated by packet relaying from one processor to neighbors. Thus, establishing a (virtual) communication link between p_s and p_r in which p_s is the sender and p_r is the receiver. We assume the communication graph is dynamic, and is constantly changed, while respecting N as the upper bound on the number of nodes in the system. Packets are exchanged by the sender and the receiver in order to deliver (high level) messages in a reliable fashion. We assume that the entire number of packets in the system

at any given time, does not exceed a known bound. We allow any churn rate, assuming that joining processors reset their own memory, and by that assist in respecting the assumed bounded packet capacity of the entire network.

The communication links are bidirectional. Namely, between every two nodes, p_i and p_j, that can exchange packets, there is a unidirectional *channel (set)* that transfers packets from p_i to p_j and another unidirectional channel that transfer packets from p_j to p_i. We model the *(communication) channel*, from node p_i to node p_j as a (non-FIFO order preserving) packet set that p_i has sent to p_j and p_j is about to receive. When p_i sends a packet m to p_j, the operation *send* inserts a copy of m to the channel from p_i to p_j as long as the upper bound of packets in the channel is respected. Once m arrives, p_j triggers the *receive* event and m is deleted from the set. The communication channel is non-FIFO and has no reliability guarantees. Thus, at any time the sent packets may be omitted, reordered, and duplicated, as long as the link capacity bound is not violated. We note that transient faults can bring the system to consist of arbitrary, and yet capacity bounded, channel sets from which convergence should start. We assume that when node p_i sends a packet, *pckt*, infinitely often through the communication link from p_i to p_j, p_j receives *pckt* infinitely often. We intentionally do not specify (the possible unreliable) routing scheme that is used to forward a packet from the sender to the receiver, e.g., flooding, shortest path routing. We assume that the overall network capacity allows a channel from p_i to p_j to contain at most *capacity* packets at any time, where *capacity* is a known constant. However, it should be noted that although the channel has a maximal capacity, packets in the channel may be duplicated infinitely many times because even if the channel is full, packets in the channel may be either lost or received. This leaves places for other packets to be (infinitely often) duplicated and received by p_j.

Self-stabilizing algorithms do not terminate (see [8]). 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*. Where a step starts with local computations and ends with a single communication operation, which is either *send* or *receive* of a packet. 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 packet or a periodic timer going off triggering p_i to send. Note that the system is totally asynchronous and the non-fixed spontaneous send of nodes and 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. The execution of an algorithm step can change the node state. 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 packets in transit for p_i). We define an *execution (or run)* $R =$

$c[0], a[0], c[1], a[1], \ldots$ 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 is *fair* if every node, p_i, executes infinitely many steps in R. We represent the omissions, duplications and reordering using environment steps that are interleaved with the steps of the processors in the run R. In every fair run, the environment steps do not prevent communication, namely, infinite *send* operations of p_i of a packet, *pckt*, to p_j implies infinite *receive* operations of *pckt* by p_j.

The system is asynchronous and the notion of time, for example, when considering system convergence to legal behavior, is measured by the number of *asynchronous rounds*, where the first asynchronous round is the minimal prefix of the execution in which every node sends at least one packet to every neighbor and one of these packets is received by each neighbor. Thus, we nullify the infinite power of omissions, duplications and reordering when measuring the algorithm performance. Moreover, we ensure that packets sent are eventually received; otherwise the channel is, in fact, disconnected. The second asynchronous round is the first asynchronous round in the suffix of the run that follows the first asynchronous round, and so on. We measure the communication costs by the number of packets sent in synchronous execution in which each packet sent by p_s arrives to its destination, p_r, in one time unit, and before p_s sends any additional packet to p_r.

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 (unbounded) execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

The *self-stabilizing end-to-end communication* (S^2E^2C) algorithm provides FIFO guarantee for bounded networks that omit duplicate and reorder packets. Moreover, the algorithm considers arbitrary starting configurations and ensures error-free message delivery. In detail, given a system's execution R, and a pair, p_s and p_r, of sending and receiving nodes, we associate the message sequences $s_R = m_0, m_1, m_2, \ldots$, of messages fetched by p_s, with the message sequence $r_R = m'_0, m'_1, m'_2, \ldots$ of messages delivered by p_r. Note that we list messages according to the order they are fetched (from the higher level application) by the sender, thus two or more (consecutive or non-consecutive) messages can be identical. The S^2E^2C task requires that for every legal execution $R \in LE$, there is an infinite suffix, R', in which infinitely many messages are delivered, and $s_{R'} = r_{R'}$. It should be noted that packets are not actually received by the receiver in their correct order but eventually it holds that messages are delivered by the receiver (to higher level application) in the right order.

3 The End-to-End Algorithm

Dynamic networks have to overcome a wide range of faults, such as message corruption and omission. It often happens that networking techniques, such as retransmissions and multi-path routing, which are used for increasing robustness, can cause undesirable behavior, such as message duplications and reordering. We present a self-stabilizing end-to-end communication algorithm that uses the network's bounded capacity, to cope with packet corruptions, omissions, duplications, and reordering. We abstract the entire network by two directed channels, one from the sender to the receiver and one from the receiver to the sender, where each abstract channel is of a bounded capacity. These two abstract channels can omit, reorder and duplicate packets. We regard two nodes, p_s, p_r, as sender and receiver, respectively. Sender p_s sends packets with distinct labels infinitely often until p_s receives a sufficient amount of corresponding distinct acknowledgment labels from the receiver p_r.

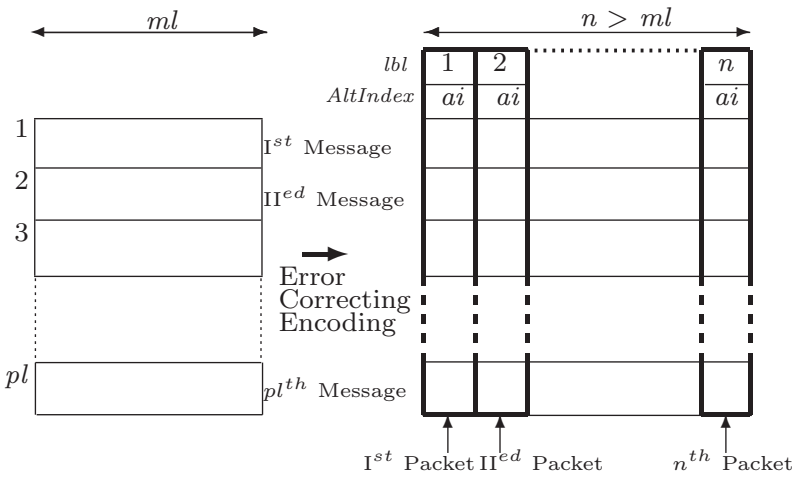

Fig. 1. Packet formation from messages

For the sake of readability, we start describing the algorithm using large overhead, before showing ways to dramatically reduce the overhead. The sender repeatedly sends each message m with a three state *alternating index*, which is either 0, 1 or 2. We choose to discuss, without the loss of generality, the case of a message with alternating index 0, where $\langle 0, m\rangle$ is repeatedly sent in $(2 \cdot \mathit{capacity} + 1)$ packet types. Each type uses a distinct label in the range 1 to twice the capacity plus 1. Namely, the types are: $\langle 0, 1, m\rangle$, $\langle 0, 2, m\rangle$, ..., $\langle 0, 2 \cdot \mathit{capacity} + 1, m\rangle$. The sender waits for an acknowledgment of the packet arrival for each of the $(2 \cdot \mathit{capacity} + 1)$ distinct labels, and an indication that the receiver delivered a message due to the arrival of $(\mathit{capacity}+1)$ packets with alternating index 0. The receiver accumulates the arriving packets in an array of $(2 \cdot \mathit{capacity}+1)$ entries, where each entry, j, stores the last arriving packet with distinct label j. Whenever the receiver finds that $(\mathit{capacity} + 1)$ recorded array entries share the same alternating index, for example 1, the receiver delivers the message m encapsulated in one in-coming packet recorded in the array – this packet has the alternating index of the majority of recorded packets; 1 in our example. Then, the receiver resets its array and starts accumulating packets again, until $(\mathit{capacity} + 1)$ recorded copies, with the same alternating index reappear. The receiver always remembers the last delivered alternating index, ldai, that caused the reset of its array, and does not deliver two successive messages with the same alternating index. Each packet $\langle \mathit{ai}, \mathit{lbl}, m\rangle$ that arrives

to the receiver is acknowledged by $\langle lbl, ldai \rangle$. The sender accumulates the arriving packet in an array of $(2 \cdot capacity + 1)$ entries and waits to receive a packet for each entry, and to have a value of $ldai$ that is equal to the alternating index the sender is currently using in the sent packets in at least $(capacity + 1)$ of the recorded packets. Once such a packet set arrives, the sender resets its array, fetches a new message, m', to be delivered, and increments the alternating index by 1 modulo 3 for the transmission process of the next message, m'.

The correctness considers the fact that the receiver always acknowledges incoming packets, and hence the sender will infinitely often fetch messages. Following the first fetch of the sender, the receiver follows the sender's alternating index, records it in $ldai$, and acknowledges this fact. We consider an execution in which the sender changes the alternating index in to x, $x+1$, $x+2$, x (all modulo 3). In this execution, the sender is acknowledged that the receiver changes $ldai$ to $x+1$ and then to $x+2$, while the sender does not send packets with alternating index x, thus, the last x delivery in the sequence must be due to fresh packets, packets sent after the packets with alternating index $x+2$ were sent, and cause a delivery.

In the preceding text a simplified algorithm with a large overhead was presented – a more efficient algorithm is described in the following. The basic idea is to enlarge the arrays to have more than $n > (2 \cdot capacity + 1)$ recorded packets. Roughly speaking, in such a case the minority of the distinct label packets accumulated in the arrays are erroneous, i.e., packet copies that were accumulated in the network prior to the current fetch (maximum $capacity$). The other $n - capacity$ distinct label accumulated packets are correct. Thus, as we know the maximal amount of unrelated packets, we can manipulate the data so that the $n - capacity$ correct packets, each of length pl will encode, by means of error correcting codes, pl messages each of length ml, a length slightly shorter than n. The sender fetches a window of pl messages each of length ml, where pl is the maximal packet length beyond the header. The sender then uses error correcting codes so that a message of length ml is coded by a word of length n such that the encoded word can bare up to $capacity$ erroneous bits. The pl encoded messages of length n are then converted to n packets of length pl in a way that the i^{th} message out of the ml fetched messages is encoded by the i^{th} bits of all the n distinct packets that are about to be transmitted. So eventually, the first bit of all distinct labeled packets, ordered by their distinct labels, encode, with redundancy, the first message, and the second bit of all distinct labeled packets, encode, with redundancy, the second message, etc. Fig. 1 shows the formation of the n packets from the pl messages. When the receiver accumulates n distinct label packets, the $capacity$ of the packets may be erroneous. However, since the i^{th} packet, out of the n distinct packets, encodes the i^{th} bits of all the pl encoded messages, if the i^{th} packet is erroneous, then the receiver can still decode the data of the original pl messages each of length $ml < n$. The i^{th} bit in each encoded message may be wrong, in fact, capacity of packets maybe erroneous yielding capacity of bits that may be wrong in each encoded message, however, due to the error correction, all the original pl messages of length ml can

be recovered, so the receiver can deliver the correct pl messages in the correct order.

In this case, the sender repeatedly sends n distinct packets and the receiver keeps sending $(capacity + 1)$ packets each with a distinct label in the range 1 to $(capacity + 1)$. In addition, each of these packets contains the receiver's current value of $ldai$. The packets from the receiver are sent infinitely often, not necessarily as a response to its received packets. When the receiver accumulates n distinct label packets with the same alternating index, it recovers the original pl messages, delivers them, resets its received packets array and changes its $ldai$ to the alternating index of the packets that it just delivered. We note that these received packets must be different from its current $ldai$ because the receiver does not accumulate packets if their alternating index is equal to its current $ldai$. The sender may continue sending the n packets with alternating index $ldai$, until the sender accumulates $(capacity + 1)$ distinct label acknowledging packets with alternating index $ldai$. However, because now the packets' alternating index is equal to its current $ldai$, the receiver does not accumulate them, and hence does not deliver a duplicate. Once the sender accumulates $(capacity+1)$ packets with $ldai$ equal to its alternating index, it will fetch pl new messages, encode and convert them to n distinct label packets and increase its alternating index by 1 modulo 3.

The correctness arguments use the same facts mentioned above in the majority based algorithm. Eventually, we will reach an execution in which the sender fetches a new set of messages infinitely often and the receiver will deliver the messages fetched by the sender before the sender fetches the next set of messages. Eventually, every set of pl fetched messages is delivered exactly once because after delivery the receiver resets its packets record array and changes $ldai$ to be equal to the senders alternating index. The receiver stops accumulating packets from the sender until the sender fetches new messages and starts sending packets with a new alternating index. Between two delivery events of the receiver, the receiver will accumulate n distinct label packets of an identical alternating index, where $n - capacity$ of them must be fetched by the sender after the last delivery of messages by the receiver. The fact, which reflects such behavior at the receiver node, is that the sender only fetches new messages after it gets $capacity + 1$ distinct packets with $ldai$ equal to its current alternating index. When the receiver holds n distinct label packets with maximum capacity erroneous packets, it can convert the packets back to the original messages by applying the error correction code capabilities and deliver the original message correctly.

Algorithm Description Algorithms 1 and 2 implement the proposed S^2E^2C sender-side and receiver-side algorithms, respectively. The two nodes, p_s and p_r, are the sender and the receiver nodes respectively. The Sender algorithm consists of a do forever loop statement (line 2 of the Sender algorithm), where the sender, p_s, assures that all the data structures comprises only valid contents. I.e., p_s checks that the ACK_set_s holds packets with alternating index equal to the senders current $AltIndex_s$ and the labels are between 1 and $capacity + 1$.

Algorithm 1: Self-Stabilizing End-to-End Algorithm (Sender)

Persistent variables:
$AltIndex$: an integer $\in [0, 2]$ that states the current alternating index value
ACK_set: at most $capacity + 1$ acknowledgment set, where items contain labels and last delivered alternating indexes, $\langle lbl, ldai \rangle$
$packet\_set$: n packets, $\langle AltIndex, lbl, dat \rangle$, to be sent, where $lbl \in [1, n]$ and dat is data of size pl bits

Interface:
$Fetch(NumOfMessages)$ Fetches $NumOfMessages$ messages from the application and returns them in an array of size $NumOfMessages$ according to their original order
$Encode(Messages[])$ receives an array of messages of length ml each, M, and returns a message array of identical size M', where message $M'[i]$ is the encoded original $M[i]$, the final length of the returned $M'[i]$ is n and the code can bare $capacity$ mistakes

```
1  Do forever begin
2      if (ACK_set ⊈ {AltIndex} × [1, capacity + 1]) then
       (ACK_set, messages) ← (∅, Encode(Fetch(pl)))
3      foreach pckt ∈ packet_set() do send pckt
4  Upon receiving ACK = ⟨lbl, ldai⟩ begin
5      if lbl ∈ [1, capacity + 1] ∧ ldai = AltIndex then
6          ACK_set ← ACK_set ∪ {ACK}
7          if capacity <| ACK_set | then begin
8              AltIndex ← (AltIndex + 1) mod 3
9              (ACK_set, messages) ← (∅, Encode(Fetch(pl)))
10 Function packet_set() begin
11     foreach (i, j) ∈ [1, n] × [1, pl] do let data[i].bit[j] = messages[j].bit[i]
12     return {⟨AltIndex, i, data[i]⟩}_{i∈[1,n]}
```

In case any of these conditions is unfulfilled, the sender resets its data structures (line 2 of the Sender algorithm). Subsequently, p_s triggers the *Fetch* and the *Encode* interfaces (line 2 of the Sender algorithm). Before sending the packets, p_s executes the $packet\_set()$ function (line 3 of the Sender algorithm).

The Sender algorithm, also, handles the reception of acknowledgments $ACK_s = \langle lbl, ldai \rangle$ (line 4 of the Sender algorithm). Each ACK_s has distinct labels, corresponding to already transmitted packets. On the reception of the $capacity + 1$ distinct label ACK_s, p_s keeps ACK_s in ACK_set_s (line 6 of the Sender algorithm), if ACK_s have the value of $ldai$ (last delivered alternating index) equals to $AltInex$ (line 5 of the Sender algorithm). When p_s gets an ACK_s packet ($capacity + 1$) times (line 7 of the Sender algorithm), p_s changes $AltIndex_s$ (line 8 of the Sender algorithm). Afterwards, p_s does reset ACK_set_s and calls $Fetch()$ and $Encode()$ interfaces (line 9 of the Sender algorithm).

Algorithm 2: Self-Stabilizing End-to-End Algorithm (Receiver)

Persistent variables:
$packet\_set$: packets, $\langle AltIndex, lbl, dat\rangle$, received, where $label \in [1, n]$ and dat is data of size pl bits
$LastDeliveredIndex$: an integer $\in [0, 2]$ that states the alternating index value of the last delivered packets

Interface:
$Decode(Messages[])$ receives an array of encoded messages, M', of length n each, and returns an array of decoded messages of length ml, M, where $M[i]$ is the decoded $M'[i]$. The code is the same error correction coded by the sender and can correct up to $capacity$ mistakes
$Deliver(messages[])$ receives an array of messages and delivers them to the application by the order in the array

Macros:
$P(ind) = \{\langle ind, *, *\rangle \in packet\_set\}$

```
1  Do forever begin
2      if {⟨ai, lbl⟩ : ⟨ai, lbl, ∗⟩ ∈ packet_set} ⊈
       {[0, 2] \ {LastDeliveredIndex}} × [1, n] × {∗}∨
       (∃⟨ai, lbl, dat⟩ ∈ packet_set : ⟨ai, lbl, ∗⟩ ∈ packet_set \ {⟨ai, lbl, dat⟩})∨
       (∃pckt = ⟨∗, ∗, data⟩ ∈ packet_set :| pckt.data |≠ pl)∨
       1 <| {AltIndex : n ≤| {⟨AltIndex, ∗, ∗⟩ ∈ packet_set} |} | then
       packet_set ← ∅
3      foreach i ∈ [1, capacity + 1] do send ⟨lbl, LastDeliveredIndex⟩
4  Upon receiving pckt = ⟨ai, lbl, dat⟩ begin
5      if ⟨ai, lbl, ∗⟩ ∉ packet_set∧
       ⟨ai, lbl⟩ ∈ ({[0, 2] \ {LastDeliveredIndex}} × [1, n])∧ | dat |= pl then
6          packet_set ← packet_set ∪ {pckt}
7          if ∃ ! ind : ind ≠ LastDeliveredIndex ∧ n ≤| P(ind) |: P(ind) =
           {⟨ind, ∗, ∗⟩ ∈ packet_set} then
8              foreach (i, j) ∈ [1, pl] × [1, n] do
9                  let messages[i].bit[j] = data.bit[i] : ⟨ind, j, data⟩ ∈ P(ind)
10             (packet_set, LastDeliveredIndex) ← (∅, ind)
11             Deliver(Decode(messages))
```

The Receiver algorithm executes at the receiver side, p_r. The receiver p_r assures its data structure, namely, $packet\_set_r$, in do forever loop (line 2 of the Receiver algorithm). The receiver p_r audits: (i) the $packet\_set_r$ holds packets with alternating index, $ai \in [0, 2]$, except $LastDeliveredIndex_r$, labels (ℓ) between 1 and n and data of size pl; (ii) the $packet\_set_r$ holds exactly one group of ai that has at least n elements. When any of the aforementioned conditions are falsified, p_r assigns the empty set to $packet\_set_r$. In addition, p_r acknowledges p_s by $(capacity + 1)$ packets (line 3 of the Receiver algorithm).

Node p_r receives a packet $pckt_r = \langle ai, lbl, dat \rangle$, see line 4 of the Receiver algorithm. If $pckt_r$ has data (dat) in the size of pl bits and $pckt_r$ has alternating index (ai) in the range of 0 to 2, excluding the $LastDeliveredIndex$ and $pckt_r$ has a label (lbl) in the range of 1 to n (line 5 of the Receiver algorithm), p_r puts $pckt_r$ in $packet\_set_r$ (line 6 of the Receiver algorithm). When p_r gets n distinct label packets of identical ai (line 7 of the Receiver algorithm), p_r forms the message from the packets (line 9 of the Receiver algorithm). Subsequent steps include the reset of the $packet\_set_r$ data structure and change of $LastDeliveredIndex_r$ to ai (line 10 of the Receiver algorithm). Next, p_r decodes and delivers the message (line 11 of the Receiver algorithm).

Correction proof The correct packet exchange between the sender and the receiver requires coordination. The sender should wait after fetching a new message batch, i.e., executing lines 8 to 9 of the Sender algorithm, until the receiver delivers a message batch, i.e., executing line 11 of the Receiver algorithm. We describe the set of legal executions for correct packet exchange before demonstrating that the Sender and the Receiver algorithms satisfy these requirements in Theorem 1, which says that the studied algorithms implement self-stabilizing end-to-end communication (S^2E^2C) task.

Let a_{s_α} be the α^{th} time that the sender is fetching a new message batch, i.e., executing lines 8 to 9 of the Sender algorithm. Let a_{r_β} be the β^{th} time that the receiver is delivering a message batch, i.e., executing line 11 of the Receiver algorithm. With respect to the self-stabilizing end-to-end communication (S^2E^2C) task and the algorithms of the Sender and the Receiver, the legal execution set includes executions, R, that interleave the a_{s_α} and the a_{r_β} steps in a manner that matches the alternating index labels. Namely, after the occurrence of $a_{s_\alpha} \in R$ in which the sender fetches a new message batch, the step $a_{s_{\alpha+1}}$ should not occur before $a_{r_\beta} \in R$ in which the receiver delivers *that* message batch (Lemma 3). Similarly, after the occurrence of $a_{r_\beta} \in R$ in which the receiver delivers a message batch, the step $a_{r_{\beta+1}}$ should not occur before $a_{s_\alpha} \in R$ in which the sender fetches the next message batch (Lemma 4).

In addition, the a_{s_α} and the a_{r_β} steps should have matching alternating indices. The proof shows that the sender, p_s, increments its $AltIndex_s = s\_index_\alpha$ value on every a_{s_α} in a modulo 3 fashion, and the receiver, p_r, adopts s_index_α and deliver its message batch in step a_{r_β} after receiving at least $n - capacity$ packets that are tagged by s_index_α. Similarly, p_r acknowledges the received packets using the tag $LastDeliveredIndex_r = r\_index_\beta$, and then p_s proceeds to fetch a next message batch in $a_{s_{\alpha+1}}$ after receiving at least more than *capacity* acknowledgments.

We note that the proof implies that within a constant number of asynchronous rounds, the receiver, p_r, receives an entire batch of n packets from its incoming abstract channel out of which $n - capacity$ packets are from the sender, p_s. This is true because: (1) we assume that when the sender sends a packet infinitely often through the abstract channel, the receiver receives the packet infinitely often, and (2) the proof shows that the sender does not stop sending its current batch of messages, before guaranteeing that the

current message batch had arrived to the receiver, p_r, and p_r had delivered it. Moreover, analogous arguments to arguments (1) and (2) above imply the number of asynchronous rounds, in which the sender, p_s, receives an entire batch of $(capacity+1)$ acknowledgments that at least one of them is from the receiver.

Lemmas 1 and 2 are needed for the proof of lemmas 3 and 4. Throughout we refer to R as an execution of the Sender and the Receiver algorithms, where p_s executes the Sender algorithm and p_r executes the Receiver algorithm.

Lemma 1. *Let $c_{s_\alpha}(x)$ be the x^{th} configuration between a_{s_α} and $a_{s_{\alpha+1}}$ and $ACK_\alpha = \{ack_\alpha(\ell)\}_{\ell\in[1,capacity+1]}$ be a set of acknowledgment packets, where $ack_\alpha(\ell) = \langle \ell, s\_index_\alpha\rangle$.*

For any given $\alpha > 0$, there is a single index value, $s\_index_\alpha \in [0,2]$, such that for any $x > 0$, it holds that $AltIndex_s = s\_index_\alpha$ in $c_{s_\alpha}(x)$. Moreover, between a_{s_α} and $a_{s_{\alpha+1}}$ there is at least one configuration c_{r_β}, in which $LastDeliveredIndex_r = s\_index_\alpha$. Furthermore, between a_{s_α} and $a_{s_{\alpha+1}}$, the sender, p_s, receives from the channel from p_r to p_s, the entire set, ACK_α, of acknowledgment packets (each packet at least once), and between (the first) c_{r_β} and $a_{s_{\alpha+1}}$ the receiver must send at least one $ack_\alpha(\ell) \in ACK_\alpha$ packet, which p_s receives.

Proof. We start by showing that s_index_α exists before showing that c_{r_β} exists and that p_s receives ack_α from p_r between a_{s_α} and $a_{s_{\alpha+1}}$.

The value of $AltIndex_s = s\_index_\alpha$ is only changed in line 8 of the Sender algorithm. By the definition of a_{s_α}, line 8 is not executed by any step between a_{s_α} and $a_{s_{\alpha+1}}$. Therefore, for any given α, there is a single index value, $s\_index_\alpha \in [0,2]$, such that for any $x > 0$, it holds that $AltIndex_s = s\_index_\alpha$ in $c_{s_\alpha}(x)$.

We show that c_{r_β} exists by showing that, between a_{s_α} and $a_{s_{\alpha+1}}$, there is at least one acknowledge packet, $\langle lbl, ldai\rangle$, that p_r sends and p_s receives, where $ldai = s\_index_\alpha$. This proves the claim because p_r's acknowledgments are always sent with $ldai = LastDeliveredIndex_r$, see line 3.

We show that, between a_{s_α} and $a_{s_{\alpha+1}}$, the receiver p_r sends at least one of the $ack_\alpha(\ell) \in ACK_\alpha$ packets that p_s receives. We do that by showing that p_s receives, from the channel from p_r to p_s , more than *capacity* packets, i.e., the set ACK_α. Since *capacity* bounds the number of packets that, at any time, can be in the channel from p_r to p_s , at least one of the ACK_α packets, say $ack_\alpha(\ell')$, must be sent by p_r and received by p_s between a_{s_α} and $a_{s_{\alpha+1}}$. This in fact proves that p_r sends $ack_\alpha(\ell')$ after c_{r_β}.

In order to demonstrate that p_s receives the set ACK_α, we note that $ACK\_set = \emptyset$ in configuration $c_{s_\alpha}(1)$, which immediately follows a_{s_α}, see line 9 of the Sender algorithm. The sender tests the arriving acknowledgment packet, ack_α, in line 5 of the Sender algorithm. It tests ack_α's label to be in the range of $[1, capacity + 1]$, and that they are of ack_α's form. Moreover, it counts that $(capacity + 1)$ different packets are added to ACK_set by adding them to ACK_set, and not executing lines 8 to 9 of the Sender algorithm before at least $(capacity + 1)$ distinct packets are in ACK_set.

Lemma 2 (proof appears in [10]). *Let $c_{r_\beta}(y)$ be the y^{th} configuration between a_{r_β} and $a_{r_{\beta+1}}$, and $PACKET_\beta(r\_index'_\beta) = \{packet_\beta(\ell, r\_index'_\beta)\}_{\ell\in[1,n]}$ be a packet set, where $packet_{\beta,r\_index'_\beta}(\ell) = \langle r\_index'_\beta, \ell, * \rangle$.*

For any given $\beta > 0$, there is a single index value, $r\_index_\beta \in [0,2]$, such that for any $y > 0$, it holds that $LastDeliveredIndex_r = r\_index_\beta$ in configuration $c_{r_\beta}(y)$. Moreover, between a_{r_β} and $a_{r_{\beta+1}}$ there is at least one configuration, c_{s_α}, such that $AltIndex_s \neq r\_index_\beta$. Furthermore, there exists a single $r\_index'_\beta \in [0,2] \setminus \{r\_index_\beta\}$, such that the receiver, p_r, receives all the packets in $PACKET_\beta(r\_index'_\beta)$ at least once between c_{s_α} and $a_{r_{\beta+1}}$, where at least $n - capacity > 0$ of them are sent by the sender p_s between a_{r_β} and $a_{r_{\beta+1}}$.

Lemmas 3 and 4 borrow their notations from lemmas 1 and 2. Lemma 4 shows that between a_{s_α} and $a_{s_{\alpha+1}}$, there is exactly one a_{r_β} step.

Lemma 3. *Between a_{s_α} and $a_{s_{\alpha+1}}$, the receiver takes exactly one a_{r_β} step, and that between a_{r_β}, and $a_{r_{\beta+1}}$, the sender takes exactly one $a_{s_{\alpha+1}}$ step.*

Proof. We start by showing that between a_{s_α} and $a_{s_{\alpha+1}}$, there is at least one a_{r_β} step before showing that there is exactly one such a_{r_β} step when $\alpha > 2$. Then, we consider a proof for showing that between a_{r_β} and $a_{r_{\beta+1}}$, there is at least one a_{s_α} step before showing that between a_{r_β} and $a_{r_{\beta+1}}$, there is exactly one a_{s_α} step when $\beta > 2$.

By Lemma 1 and line 8 of the Sender algorithm, in any configuration, $c_{s_1}(x)$, that is between a_{s_1} and a_{s_2}, the sender is using a single alternating index, s_index_1, and in any configuration, $c_{s_2}(x)$, that is between a_{s_2} and a_{s_3}, the sender is using a single alternating index, s_index_2, such that $s\_index_2 = s\_index_1 + 1 \bmod 3$. In a similar manner, we consider configuration, $c_{s_\alpha}(x)$, that is between a_{s_α} and $a_{s_{\alpha+1}}$ and conclude Equation (**??**).

Lemma 1 also shows that for $\alpha \in (1, 2, \ldots)$, there are configurations, c_{r_α}, in which $LastDeliveredIndex_r = s\_index_\alpha$. This implies that between a_{s_α} and $a_{s_{\alpha+1}}$, the receiver changes the value of $LastDeliveredIndex_r$ at least once, where $\alpha \in (1, 2, \ldots)$. Thus, by a_{r_β}'s definition and line 10 of the Receiver algorithm, there is at least one a_{r_β} step between a_{s_α} and $a_{s_{\alpha+1}}$.

To see that when $\alpha > 2$ there is exactly one such a_{r_β} step between a_{s_α} and $a_{s_{\alpha+1}}$, we consider the case in which between a_{s_α} and $a_{s_{\alpha+1}}$, there are several a_{r_β} steps, i.e., $a_{r_{\beta_{first}}}, \ldots, a_{r_{\beta_{last}}}$. In particular we consider the $a_{s_{\alpha-1}}$, $a_{r_{\beta-1_{last}}}$, a_{s_α}, $a_{r_{\beta_{first}}}$, $a_{r_{\beta_{last}}}$, $a_{s_{\alpha+1}}$ steps and show that $a_{r_{\beta+1_{first}}} = a_{r_{\beta+1_{last}}}$. Let us assume, in the way of a proof by contradictions that $a_{r_{\beta+1_{first}}} \neq a_{r_{\beta+1_{last}}}$. We show that there is an $a_{s_{\alpha'}}$ step between $a_{r_{\beta+1_{first}}}$ and $a_{r_{\beta+1_{last}}}$.

By Lemma 2, between $a_{r_{\beta_{first}}}$ and $a_{r_{\beta_{last}}}$, there is at least one configuration, $c_{s_{\alpha'}}(x)$, for which $AltIndex_s \neq r\_index_{\beta-1_{last}}$, and at least one configuration, $c_{s_{\alpha''}}(x)$, for which $AltIndex_s \neq r\_index_{\beta+1_{first}}$.

Suppose that $\alpha' = \alpha''$. By a_{s_α}'s definition, line 3 of the Sender algorithm and the function *packet_set*(), the sender changes $AltIndex_s$'s value in step $a_{s_{\alpha'}}$ that occurs between $a_{r_{\beta+1_{first}}}$ and $a_{r_{\beta+1_{last}}}$. For the case of $\alpha' \neq \alpha''$, we use similar arguments and consider the sequence of all $c_{s_{\alpha'}}(x), c_{s_{\alpha''}}(x), \ldots$

configurations between $a_{r_{\beta_{first}}}$ and $a_{r_{\beta_{last}}}$ and their corresponding $AltIndex_s$'s values. By similar arguments to the case of $\alpha' = \alpha''$, any consecutive pair of $AltIndex_s$ implies the existence of an a_{s_α} between $a_{r_{\beta_{first}}}$ and $a_{r_{\beta_{last}}}$. Thus, a contradiction.

Lemma 4 shows that between a_{r_β} and $a_{r_{\beta+1}}$, there is exactly one a_{s_α} step, and its proof follows similar arguments as the ones in Lemma 3.

Lemma 4 (proof appears in [10]). *Between a_{r_β} and $a_{r_{\beta+1}}$, the sender takes exactly one $a_{s_{\alpha+1}}$ step.*

Lemmas 3 and 4 facilitates the proof of Theorem 1.

Theorem 1 (S^2E^2C). *Within a constant number of asynchronous rounds, the system reaches a safe configuration (from which a legal execution starts). Moreover, following a safe configuration, Algorithm 2 delivers every new sent message batch within a constant number of asynchronous rounds.*

4 Conclusions

Self-stabilizing end-to-end data communication algorithms for bounded capacity dynamic networks have been presented in this extended abstract. The proposed algorithms inculcate error correction techniques for the delivery of messages to their destination without omissions, duplications or reordering. We consider two nodes, one as the sender and the other as the receiver. In many cases, however, two communicating nodes may act both as senders and receivers simultaneously. In such situations, acknowledgment piggybacking may reduce the overhead needed to cope with the capacity irrelevant packets that exist in each direction, from the sender to the receiver *and* from the receiver to the sender. Using piggybacking, the overhead is similar in both directions. The obtained overhead is proportional to the ratio between the number of bits in the original message, and the number of bits in the coded message, which is a code that withstand *capacity* corruptions. Thus, for a specific *capacity*, assuming the usage of efficient encoding, the overhead becomes smaller as the message length grows.

References

1. Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.
2. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS*, pages 268–277. IEEE Computer Society, 1991.
3. A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In A. Arora, editor, *Workshop on Self-stabilizing Systems (ICDCS'99)*, pages 78–85. IEEE Computer Society, 1999.
4. H. C. Chung, P. Robinson, and J. L. Welch. Brief announcement: Regional consecutive leader election in mobile ad-hoc networks. In *6th International Workshop on Algorithms for Sensor Systems*, pages 89–91, 2010.

5. A. Cournier, S. Dubois, and V. Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed (IPDPS'09)*, pages 1–11, 2009.
6. A. K. Datta, L. L. Larmore, and H. Piniganti. Self-stabilizing leader election in dynamic networks. In *12th International Symposium Stabilization, Safety, and Security of Distributed Systems SSS'10*, pages 35–49, 2010.
7. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
8. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
9. S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.
10. S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing data link over non-fifo channels without duplication. Technical Report 2012:01, Chalmers University of Technology, 2012. ISSN 1652-926X.
11. S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997. Conference version PODC 1991: 281-293.
12. S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilitzing group communication in ad hoc networks. In *PODC*, page 259, 2002.
13. S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 70–79, 2002.
14. S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mob. Comput.*, 5(7):893–905, 2006.
15. S. Dolev and J. L. Welch. Crash resilient communication in dynamic networks. *IEEE Trans. Computers*, 46(1):14–26, 1997.
16. O. Flauzac and V. Villain. An implementable dynamic automatic self-stabilizing protocol. In *ISPAN*, pages 91–97. IEEE Computer Society, 1997.
17. M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.
18. B. Haeupler and D. R. Karger. Faster information dissemination in dynamic networks via network coding. In *30th Annual ACM Symposium on Principles of Distributed Computing (PODC'11)*, pages 381–390, 2011.
19. R. Ingram, P. Shields, J. E. Walter, and J. L. Welch. An asynchronous leader election algorithm for dynamic networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*, pages 1–12, 2009.
20. M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.
21. F. Kuhn, T. Locher, and R. Oshman. Gradient clock synchronization in dynamic networks. *Theory Comput. Syst.*, 49(4):781–816, 2011.
22. F. Kuhn, N. A. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *ACM Symposium on Theory of Computing (STOC'10)*, pages 513–522, 2010.
23. F. Kuhn, R. Oshman, and Y. Moses. Coordinated consensus in dynamic networks. In *30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, pages 1–10, 2011.
24. J. Spinelli. Self-stabilizing sliding window arq protocols. *IEEE/ACM Trans. Netw.*, 5(2):245–254, 1997.
25. A. S. Tanenbaum. *Computer networks (4. ed.).* Prentice Hall, 2002.