Disconnected Agreement in Networks Prone to Link Failures

Bogdan S. Chlebus[1], Dariusz R. Kowalski[1⋆], Jan Olkowski[2], and Jędrzej Olkowski[3]

[1] School of Computer and Cyber Sciences, Augusta University, Georgia, USA
[2] University of Maryland, College Park, Maryland, USA
[3] Wydział Matematyki, Mechaniki i Informatyki, Uniwersytet Warszawski, Warszawa, Poland

Abstract. We consider deterministic distributed algorithms for reaching agreement in synchronous networks of arbitrary topologies. Links are bi-directional and prone to failures while nodes stay non-faulty at all times. A faulty link may omit messages. Agreement among nodes is understood as holding in each connected component of a network obtained by removing faulty links – we call it a "disconnected agreement". We introduce the concept of stretch, which is the number of connected components of a network, obtained by removing faulty links, minus 1 plus the sum of diameters of connected components. We define the concepts of "fast" and "early-stopping" algorithms for disconnected agreement by referring to stretch. We consider trade-offs between the knowledge of nodes, the size of messages, and the running times of algorithms. A network has n nodes and m links. We give a general disconnected agreement algorithm operating in $n+1$ rounds that uses messages of $\mathcal{O}(\log n)$ bits. Let λ be an unknown stretch occurring in an execution; we give an algorithm working in time $(\lambda+2)^3$ and using messages of $\mathcal{O}(n \log n)$ bits. We show that disconnected agreement can be solved in the optimal $\mathcal{O}(\lambda)$ time, but at the cost of increasing message size to $\mathcal{O}(m \log n)$. We also design an algorithm that uses only $\mathcal{O}(n)$ non-faulty links and works in time $\mathcal{O}(nm)$, while nodes start with their ports mapped to neighbors and messages carry $\mathcal{O}(m \log n)$ bits. We prove lower bounds on the performance of disconnected-agreement solutions that refer to the parameters of evolving network topologies and the knowledge available to nodes.

Keywords: Network · Synchrony · Omission link failures · Agreement · Time complexity · Message size · Link use

1 Introduction

We introduce a variant of agreement and present deterministic distributed algorithms for this problem in synchronous networks. Nodes represent processing units and links model bi-directional communication channels between pairs of nodes. Links are prone to failures but nodes stay operational at all times. A

⋆ This work is partially supported by the NSF grant number 2131538.

algorithm	time	message size	# links	knowledge	lower bound
Fast-Agreement	Λ †	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$	Λ known	time $\lambda \leq \Lambda$
SM-Agreement	$n+1$	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$	minimal	time λ
LM-Agreement	$(\lambda+2)^3$	$\mathcal{O}(n \log n)$	$\mathcal{O}(m)$	minimal	time λ
ES-Agreement	$\lambda+2$ †	$\mathcal{O}(m \log n)$	$\mathcal{O}(m)$	minimal	time λ
OL-Agreement	$\mathcal{O}(nm)$	$\mathcal{O}(m \log n)$	$2n$ †	neighbors known	# links $\Omega(n)$

Table 1. A summary of the given deterministic distributed algorithms for disconnected agreement and their respective performance bounds. The dagger symbol † indicates the asymptotic optimality of the respective upper bound.

faulty link may not convey a message transmitted at a round. A link that has omitted a message manifested its faultiness and is considered *unreliable* until the end of the execution. We model a network with link failures as evolving through a chain of sub-networks, obtained by removing unreliable links.

We study agreement that allows nodes in different connected components of the network, obtained by removing unreliable links, to decide on different values but still requires nodes within a connected component to decide on the same value.

We use a network's dynamic attribute, called "stretch", which is an integer determined by the number of connected components and their diameters (see Section 2 for formal definition). The purpose of using stretch is to consider scalability of disconnected agreement solutions to networks evolving through link failures.

A summary of the results. We introduce the problem of disconnected agreement and give deterministic algorithms for this problem in synchronous networks with links prone to failures. Let n denote the number of nodes and m the number of links in an initial network. An upper bound on stretch, denoted Λ, could be given to all nodes, with an understanding that faults occurring in an execution are restricted such that the actual stretch never surpasses Λ. An algorithm solving disconnected agreement with a known upper bound Λ on the stretch is considered "fast" if it runs in time $\mathcal{O}(\Lambda)$. A fast solution to disconnected agreement is discussed in Section 3. We also show a lower bound which demonstrates that, for each natural number λ and an algorithm solving disconnected agreement in networks prone to link failures, there exists a network that has stretch λ and such that each execution of the algorithm on this network takes at least λ rounds. In Section 4, we show how to solve disconnected agreement in $n+1$ rounds with short messages of $\mathcal{O}(\log n)$ bits in networks where nodes have minimal knowledge. We give an algorithm relying on minimal knowledge working in time $(\lambda+2)^3$ and using linear messages[†] of $\mathcal{O}(n \log n)$ bits, where λ is an un-

[†] We call a message 'linear' if it could carry at most $O(n)$ ids (each id has $O(\log n)$ bits).

known stretch occurring in an execution; this algorithm is presented in Section 5. A disconnected agreement solution is considered "early-stopping" if it operates in time proportional to the unknown stretch actually occurring in an execution. In Section 6, we develop an early-stopping solution to disconnected agreement relying on minimal knowledge that employs messages of $\mathcal{O}(m \log n)$ bits. We propose to count the number of reliable links used by a communication algorithm during its execution as its performance metric. To make this performance measure meaningful, the nodes need to start knowing their neighbors, in having a correct mapping of communication ports to neighbors. In Section 7, we give a solution to disconnected agreement that uses at most an asymptotically optimum number $2n$ of reliable links and works in $\mathcal{O}(nm)$ rounds, without knowing the size n of the network. We then show a separation result in Section 7: if the nodes start with their ports not mapped on neighbors, then any disconnected agreement solution has to use $\Omega(m)$ links in some networks of $\Theta(m)$ links, for all numbers n and m such that $n \leq m \leq n^2$. A summary of algorithms with their performance bounds and optimality is in Table 1. Full paper is available in [8].

The previous work on agreement in networks. Dolev [10] studied Byzantine consensus in networks with faulty nodes and gave connectivity conditions sufficient and necessary for a solution to exist; see also Fischer et al. [11], and Hadzilacos [12]. Khan et al. [13] considered a related problem in the model with restricted Byzantine faults, in particular, in the model requiring a node to broadcast identical messages to all neighbors at a round. Tseng and Vaidya [20] presented necessary and sufficient conditions for the solvability of consensus in directed graphs under the models of crash and Byzantine failures. For recent advancements, we refer the reader to [4, 7, 9, 19, 21, 22].

Next, we discuss previous work on solving consensus in networks undergoing topology changes, malfunctioning links and transmission failures.

Kuhn et al. [15] considered Δ-coordinated binary consensus in undirected graphs, whose topology could change arbitrarily from round to round, as long it stayed connected; here Δ is a parameter that bounds from above the difference in times of termination for any two nodes. Paper [15] showed how to solve Δ-coordinated binary consensus in $\mathcal{O}(\frac{nD}{D+\Delta} + \Delta)$ rounds using message of $\mathcal{O}(m^2 \log n)$ size without a prior knowledge of the network's diameter D. Comparing to our work, the paper [15] assumes that network connectivity is maintained and the Δ-coordination property imposes additional constrains on the algorithms.

Biely et al. [2] considered reaching agreement and k-set agreement in networks when communication is modeled by directed-graph topologies controlled by adversaries, with the goal to identify constraints on adversaries to make the considered problems solvable. Paper [2] solved k-set agreement in time $\mathcal{O}(3D + H)$ and using messages of $\mathcal{O}(nD \log n)$ size, where D denotes the dynamic source diameter and H denotes the dynamic graph depth, and the code of algorithm includes D. Some of our solutions can be faster and use smaller messages in this setting, since $D = E = \Lambda \geq \lambda$; for example, in dynamic networks in which $Dn = \omega(m)$.

Kuhn et al. [14] considered dynamic networks in which the network topology changes from round to round such that in every $T \geq 1$ consecutive rounds there exists a stable connected spanning subgraph, where T is a parameter. Paper [14] gave an algorithm that implements any computable function of the initial inputs, working in $\mathcal{O}(n+n^2/T)$ time with messages of $\mathcal{O}(\log n+d)$ size, where d denotes the size of input values. That solution is similar to our $\mathcal{O}(n)$ time algorithm, but it assumes the existence of a spanning connected subgraph throughout an execution, and T must be $\Omega(n)$ to result in time $\mathcal{O}(n)$, while our algorithm adjusts to disjoint connected components as they occur.

Other related work includes: agreement in complete networks in the presence of dynamic transmission failures, cf., [6, 17, 16]; almost-everywhere agreement [1]; approximate consensus [5]; and other models with transient failures [3, 18].

2 Preliminaries

We model distributed systems as collections of nodes that communicate through a wired communication network. Executions of distributed algorithms are synchronous, in that they are partitioned into global rounds coordinated across the whole network. There are n nodes in a network. Each node has a unique *name* used to determine its identity; a name can be encoded by $\mathcal{O}(\log n)$ bits.

Links connecting pairs of nodes serve as bi-directional communication channels. If at least one message is transmitted by a link in an execution then this link is *used* and otherwise it is *unused* in this execution. A link may fail to deliver a message transmitted through it at a round; once such omission happens for a link, it is considered *unreliable*. The functionality of an unreliable link is unpredictable, in that it may either deliver a transmitted message or fail to do it. A link that has never failed to deliver a message by a given round is *reliable* at this round. A path in the network is *reliable* at a round if it consists only of links that are reliable at this round. Nodes and links of a network can be interpreted as a simple graph, with nodes serving as vertices and links as undirected edges. A network at the start of an execution is represented by some *initial graph* G, which is simple and connected. An edge representing an unreliable link is removed from the graph G at the first round it fails to deliver a transmitted message. A graph representing the network evolves through a sequence of its sub-graphs and may become partitioned into multiple connected components. Once an algorithm's execution halts, we stop this evolution of the initial graph G. An evolving network, and its graph representation G, at the first round after all the nodes have halted in an execution is denoted by G_F.

We precisely define the algorithmic problem of interest as follows. Each node p starts with an initial value $\texttt{input}_p$. We assume two properties of such input values. One is that an input value can be represented by $\mathcal{O}(\log n)$ bits. The other is that input values can be compared, in the sense of belonging to a domain with a total order. In particular, finitely many initial input values contain a maximum one. We say that a node *decides* when it produces an output by setting a dedicated variable to a decision value. The operation of deciding is irrevocable. An algorithm *solves disconnected agreement* in networks with links prone to failures if the following three properties hold in all executions:

Termination: every node eventually decides.
Validity: each decision value is among the input values.
Agreement: when a node p decides then its decision value is the same as these of the nodes that have already decided and to which p is connected by a reliable path at the round of deciding.

If a message sent by a node executing a disconnected agreement solution carries a constant number of node names and a constant number of input values then the size of such a message is $\mathcal{O}(\log n)$ bits, due to our assumptions about encoding names and input values. Messages of $\mathcal{O}(\log n)$ bits are called *short*. If a message carries $\mathcal{O}(n)$ node names and $\mathcal{O}(n)$ input values then the size of such a message is $\mathcal{O}(n \log n)$ bits. We call messages of $\mathcal{O}(n \log n)$ bits *linear*.

Let H be a simple graph. If H is connected then $\operatorname{diam}(H)$ denotes the diameter of H. Suppose H has k connected components $C_1, \ldots, C_k$, where $k \geq 1$, and let $d_i = \operatorname{diam}(C_i)$ be the diameter of component C_i. The *stretch of H* is defined as a number $k - 1 + \sum_{i=1}^{k} d_i$. The stretch of a connected graph equals its diameter, because then $k = 1$. The stretch of H can be interpreted as the maximum diameter of a graph obtained from H by adding $k - 1$ edges such that the obtained graph is connected. The maximum stretch of a graph with n vertices is $n - 1$, which occurs when every vertex is isolated or, more generally, when each connected component is a line of nodes.

We say that an algorithm relies on *minimal knowledge* if each node knows its unique name and can identify a port through which a message arrives and can assign a port for a message to be transmitted through.

A disconnected-agreement algorithm in a synchronous network with links prone to failures is *early stopping* if it runs in a number of rounds proportional to the unknown stretch λ actually occurring. Such an algorithm is *fast* if it runs in a number of rounds proportional to an upper bound on stretch Λ, assuming this bound is known to all the nodes.

3 Fast Agreement

We present a fast algorithm solving disconnected agreement, assuming that a bound Λ on stretch is known to all nodes. The algorithm is called Fast-Agreement; its pseudocode is given in Figure 1.

Theorem 1. *Consider an execution of algorithm* Fast-Agreement (Λ) *in a network. If the stretch of the network never gets greater than Λ then the algorithm solves disconnected agreement in Λ rounds using messages of $\mathcal{O}(\log n)$ bits.*

We next focus on lower bounds on the number of rounds of any algorithm.

Lemma 1. *For any algorithm $\mathcal{A}$ solving disconnected agreement in networks prone to link failures, and for positive integers D and $n \geq 2D$, there exists a network*

```
algorithm Fast-Agreement ($\Lambda$)

1. initialize candidate ← input$_p$
2. repeat $\Lambda$ times
       if the current value of candidate has not been sent before then
           send candidate to all neighbors
       receive messages from all neighbors
       if a value greater than candidate has just been received
           then set candidate to the maximum value just received
3. decide on candidate
```

Fig. 1. A pseudocode of algorithm Fast-Agreement for a node p. The parameter Λ represents an upper bound on stretches, which is known to all nodes.

G with n nodes and with diameter D such that some execution of $\mathcal{A}$ on G takes at least D rounds with no link failures.

Corollary 1. *For any algorithm $\mathcal{A}$ solving disconnected agreement in networks prone to link failures, and for any even positive integer n, there exists a network G with n nodes such that some execution of $\mathcal{A}$ on G takes at least $\frac{n}{2}$ rounds with no link failures.*

Theorem 2. *For any natural number $\lambda \leq \Lambda$ and an algorithm $\mathcal{A}$ solving disconnected agreement in networks prone to link failures, there exists a network G that has stretch at most λ and such that each execution of $\mathcal{A}$ on G takes at least λ rounds.*

4 General Agreement with Short Messages

We present a general disconnected-agreement algorithm using short messages of $\mathcal{O}(\log n)$ bits. Algorithm Fast-Agreement presented in Section 3, which also employs messages of $\mathcal{O}(\log n)$ bits, relies on an upper bound on stretch Λ that is a part of code, and if the actual stretch in an execution goes beyond Λ then an execution of algorithm Fast-Agreement may not be correct. We assume in this section that nodes rely on minimal knowledge only and the given algorithm is correct for arbitrary patterns of link failures and the resulting stretches. The algorithm terminates in at most $n+1$ rounds, while the number of nodes n is not known. The running time is asymptotically optimal in case there are no failures, by Corollary 1 in Section 3.

The algorithm is called SM-Agreement, its pseudocode is in Figure 2.

Theorem 3. *Algorithm* SM-Agreement *solves disconnected agreement in $n+1$ rounds relying on minimal knowledge and using short messages of $\mathcal{O}(\log n)$ bits.*

5 Agreement with Linear Messages

The goal of this section is to develop an algorithm whose running time scales well to the stretch actually occurring in an execution. We are ready to use mes-

```
algorithm SM-Agreement

1. initialize: Inputs to empty list, round $\leftarrow 1$ ; append $(\texttt{name}_p, \texttt{input}_p)$ to Inputs
2. for each port $\alpha$ do
       initialize set Channel[$\alpha$] to empty ; send $(\texttt{name}_p, \texttt{input}_p)$ through $\alpha$
3. for each port $\alpha$ do
       if a pair $(\texttt{name}_q, \texttt{input}_q)$ received through $\alpha$ then
           add $(\texttt{name}_q, \texttt{input}_q)$ to Channel[$\alpha$] ; append $(\texttt{name}_q, \texttt{input}_q)$ to Inputs
4. repeat
   (a) for each port $\alpha$ do
           if some item in Inputs is not in Channel[$\alpha$] then
               let $x$ be the first such an item ; send $x$ through $\alpha$ ; add $x$ to Channel[$\alpha$]
   (b) for each port $\alpha$ do
           if a pair $(\texttt{name}_q, \texttt{input}_q)$ was just received through $\alpha$ then
               add $(\texttt{name}_q, \texttt{input}_q)$ to Channel[$\alpha$] ; append $(\texttt{name}_q, \texttt{input}_q)$ to Inputs
   (c) round $\leftarrow$ round $+ 1$
   until round $>$ |Inputs|
5. decide on the maximum input value in Inputs
```

Fig. 2. A pseudocode for a node p. The operation of adding an item to a set is void if the item is already in the set. The operation of appending an item to a list is void if the item is already in the list. The notation |`Inputs`| means the number of items in the list `Inputs`. For a port α, the set `Channel`[α] contains pairs of the format (node's name, node's input) that the node p has either received or sent through the port α.

sages longer than short ones used in the previous sections, and will use linear messages of $\mathcal{O}(n \log n)$ bits. Nodes are to rely on the minimal knowledge only: each node knows its own name and can distinguish ports by their communication functionality. The size of linear messages imposes constrains on the design of algorithms, and the obtained algorithm is not early stopping, but its running time is polynomial in λ.

Every node maintains a counter of round numbers, incremented when a round begins. In each round, a node p generates a new timestamp r equal to the current value of the round counter, and forms a pair $(\texttt{name}_p, r)$, which we call a *timestamp pair* of node p. Such timestamp pairs are sent to the neighbors, to be forwarded through the network. Each node node p stores a timestamp pair with the latest timestamp for a node it has ever received a timestamp pair from, and sends all such pairs to the neighbors in every round. An execution of the algorithm at a node is partitioned into *epochs*, each epoch being a contiguous interval of rounds. Epochs are not coordinated among nodes, and each node governs its own epochs. The first epoch begins at round zero, and for the following epochs, the last round of an epoch is remembered in order to discern timestamp pairs sent in the following epochs. For the purpose of monitoring progress of discovering the nodes in the connected component during an epoch, each node maintains a separate collection of timestamp pairs, which we call *pairs serving the epoch*. This collection stores only timestamp pairs sent in the current epoch, a pair with

```
algorithm LM-Agreement

1. initialize: $\texttt{candidate}_p \leftarrow \texttt{input}_p$ , $\texttt{round} \leftarrow 0$, $\texttt{Timestamps} \leftarrow \emptyset$, $\texttt{Nodes} \leftarrow \perp$
2. repeat
   (a) $\texttt{epoch} \leftarrow \texttt{round}$, $\texttt{PreviousNodes} \leftarrow \texttt{Nodes}$, $\texttt{EpochTimestamps} \leftarrow \emptyset$
   (b) repeat
        i. $\texttt{round} \leftarrow \texttt{round} + 1$
       ii. add pair $(\texttt{name}_p, \texttt{round})$ to sets Timestamps and EpochTimestamps
      iii. for each port do
           A. send Timestamps and (this-is-candidate, $\texttt{candidate}_p$) through the port
           B. receive messages coming through the port
       iv. for each received pair (this-is-candidate, $x$) do
           if $x > \texttt{candidate}_p$ then assign $\texttt{candidate}_p \leftarrow x$
        v. for each received timestamp pair $(\texttt{name}_q, y)$ do
           A. add $(\texttt{name}_q, y)$ to Timestamps if this is a good update
           B. if $y > \texttt{epoch}$ then add $(\texttt{name}_q, y)$ to EpochTimestamps if this is a good
              update
   (c) until epoch stabilized at the round
   (d) set Nodes to the set of first coordinates of timestamp pairs in EpochTimestamps
3. until PreviousNodes = Nodes
4. send (this-is-decision, $\texttt{candidate}_p$) through each port
5. decide on $\texttt{candidate}_p$
```

Fig. 3. A pseudocode for a node p. Each iteration of the main repeat-loop (2) makes an epoch. Symbol $\perp$ denotes a value different from any actual set of nodes, so the initialization of Nodes to $\perp$ in line (1) guarantees execution of at least two epochs. A *good update* of a timestamp pair for a node q either adds a first such a pair for q or replaces a present pair for q with one with a greater timestamp. At each round, p checks to see if a message of the form (this-is-decision, z) has been received, and if so then p forwards this message through each port, then decides on z, and halts.

the greatest timestamp per node which originally generated the pair. The *status of a node q at a node p* during an epoch can be either absent, updated, or stale. If the node p does not have a timestamp pair for q serving the epoch then q is *absent* at p. If at a round of an epoch the node p either adds a timestamp pair serving the epoch for an absent node q or replaces a timestamp pair of a node q by a new timestamp pair with a greater timestamp than the previously held one, then q is *updated* at this round. If the node p has a timestamp pair for a node q serving the epoch but does not replace it at a round with a different timestamp pair to make it updated, then q is *stale* at this round.

We say that an epoch of a node p *stabilizes* at a round if either no new node has its status changed from absent to updated at p or no node gets its range changed at p. If an epoch stabilizes at a round, then the epoch ends. During an epoch, a node p builds a set of names of nodes from which it has received timestamp pairs serving this epoch. A similar set produced in the previous epoch is also stored. As an epoch ends, p compares the two sets. If they are equal then p stops executing epochs, decides on the maximum input value ever learned about, notifies the neighbors of the decision, and halts.

Each node p uses a variable $\texttt{candidate}_p$, which it initializes to $\texttt{input}_p$. Node p creates a pair (this-is-candidate, $\texttt{candidate}_p$), which we call a *candidate pair of p*. Nodes keep forwarding their candidate pairs to the neighbors continually. If a node p receives a candidate pair of some other node with a value x such that $x > \texttt{candidate}_p$ then p sets its $\texttt{candidate}_p$ to x. An execution concludes with deciding by performing instruction (5). Just before deciding, a node notifies the neighbors of the decision. Once a notification of a decision is received, the recipient forwards the decision to its neighbors, decides on the same value, and halts.

The variable `round` is an integer counter of rounds, which is incremented in each iteration of the inner repeat loop by executing instruction (2(b)i). The round counter is used to generate timestamps. The variable `Timestamps` stores timestamp pairs that p has received and forwards to its neighbors. The variable `EpochTimestamps` stores timestamp pairs serving the current epoch, which have been generated after the beginning of the current epoch. Each set `Timestamps` and `EpochTimestamps` stores at most one timestamp pair per node, the one with the greatest received timestamp. Each iteration of the inner repeat loop (2b) implements one round of sending and collecting messages through all the ports by executing instruction (2(b)iii). The inner repeat loop (2b) ends as soon as the epoch stays stable at a round, which is represented by condition (2c). The variable `Nodes` stores the names of nodes from which timestamp pairs serving the epoch have been received. The variable `Nodes` is calculated at the end of an epoch by instruction (2d). The set of nodes in `Nodes` at the end of an epoch is stored as `PreviousNodes` at the start of the next epoch. The main repeat loop (2) stops to be iterated as soon as the set of names of nodes stored in `Nodes` stays the same as the set stored in `PreviousNodes`, which is checked by condition (3).

Theorem 4. *Algorithm* LM-Agreement *solves disconnected agreement in* $(\lambda + 2)^3$ *rounds, relying on minimal knowledge and using* $\mathcal{O}(n \log n)$ *bit messages.*

6 Early Stopping Agreement

We give an early-stopping disconnected agreement algorithm whose running time performance $\mathcal{O}(\lambda)$ scales optimally to the stretch λ occurring in an execution by the time of halting. Nodes rely only on the minimal knowledge, similarly as in algorithms SM-Agreement (in Section 4) and LM-Agreement (in Section 5), but messages carry $\mathcal{O}(m \log n)$ bits. This size is greater than that of short messages with $\mathcal{O}(\log n)$ bits in algorithm SM-Agreement and linear messages with $\mathcal{O}(n \log n)$ bits in algorithm LM-Agreement.

The algorithm is called ES-Agreement, its pseudocode is given in Figure 4. The pseudocode refers to a number of variables that we introduce next. A set variable `Nodes` at a node p stores the names of all the nodes that the node p has ever learned about, and a set variable `Links` stores the links known by p to have transmitted messages successfully at least once, a link is represented as a set of two names of nodes at the endpoints of the link. A set variable

```
Algorithm ES-AGREEMENT

1. initialize: Nodes ← {name_p}, Inputs ← {(name_p, input_p)}, Links ← ∅,
   Unreliable ← ∅
2. for each port do
   (a) send name_p through this port
   (b) if name_q received through this port then
         i. assign name_q to the port as a name of the neighbor
        ii. add name_q to Nodes; add edge {name_p, name_q} to Links
3. while there is an unsettled node in p's connected component in the snapshot do
      for each neighbor q do
         i. send sets Nodes, Links, Unreliable, Inputs to q
        ii. if a message from q was just received then
                 update the sets Nodes, Links, Unreliable, Inputs
                         by adding new elements included in this message from q
            else add edge {name_p, name_q} to Unreliable
4. for each neighbor q do send sets Nodes, Links, Unreliable, Inputs to q
5. decide on the maximum input value at the second coordinate of a pair in Inputs
```

Fig. 4. A pseudocode for a node p. A node q is considered unsettled by p if it is in the same connected component as p, according to the snapshot at p, and there is no pair of the form $(\mathtt{name}_q, ?)$ in $\mathtt{Inputs}_p$.

`Unreliable` stores the edges representing links known to have failed. Knowledge about failures can be acquired in two ways: either directly, when a neighbor is expected to send a message at a round and no message arrives through the link, or indirectly, contained in a snapshot received from a neighbor. A node stores all known initial input values of nodes q as pairs $(\mathtt{name}_q, \mathtt{input}_q)$ in a set variable `Inputs`. The nodes keep notifying their neighbors of the values of some of their private variables during iterations of the while loop in instruction (3) in Figure 4. A node iterates this loop until all vertices in the connected component of the node are settled, which is sufficient to decide. Once a node is ready to decide, it forwards its snapshot to all the neighbors for the last time, decides on the maximum input value in some pair in `Inputs`, and halts. An execution of the algorithm starts with each node announcing its name to all its neighbors, by executing the instruction (2) in Figure 4. This allows every node to discover its neighbors and map its ports to the neighbors' names. A node does not send its input in the first round of communication. A node sends its snapshot to the neighbors for the first time at the second round, by instruction (3) in the pseudocode in Figure 4. A node p has heard of a node q if $\mathtt{name}_q$ is in the set $\mathtt{Nodes}_p$. A node p has settled node q once the pair $(\mathtt{name}_q, \mathtt{input}_q)$ is in $\mathtt{Inputs}_p$ and the node q belongs to the connected component of p according to its snapshot.

Theorem 5. *Algorithm* ES-AGREEMENT *is an early stopping solution of disconnected agreement that relies on minimal knowledge, terminates within* $\lambda + 2$ *rounds and uses messages carrying* $\mathcal{O}(m \log n)$ *bits.*

7 Optimizing Link Use

We present an algorithm solving disconnected agreement that uses the optimal number $\mathcal{O}(n)$ of links and messages of $\mathcal{O}(m \log n)$ bits. We depart from the model of minimal knowledge of the previous sections and assume that nodes know their neighbors at the outset, in having names of the corresponding neighbors associated with all their ports. We complement the algorithm by showing that using $\mathcal{O}(n)$ links is only possible when each node starts with a mapping of ports on its neighbors, because otherwise $\Omega(m)$ is a lower bound on the link use.

The general idea of the algorithm is to have nodes build their maps of the network that include the connected component of each node. An approximation of the map at a node evolves through a sequence of snapshots of the vicinity of the node. Such a snapshot helps to coordinate choosing links through which messages are sent to extend the current snapshot to a bigger one. Input values could be a part of node attributes of the vertices on such a map. A node categorizes its incident links as either passive, active or unreliable; these are exclusive categories that evolve in time. An *active* link is used to send messages through it, so a node categorizes an incident link as active once it receives a message through it. Initially, one link incident to a node is considered as active by the node, and all the remaining incident links are considered passive. A link is *passive* at a round if none of its endpoint nodes has ever attempted a transmission through this link. A node transmits through an active port at every round, unless the node decides and halts. It follows that if a node p considers a link active, which connects it to a neighbor q, then q considers the link active as well, possibly with a delay of one round. Similarly, if a node p considers a link passive, which connects it to a neighbor q, then q considers the link passive as well, possibly for one round longer than p. A node p detects a failure of an active link and begins to consider it unreliable after the link fails to deliver a message to p as it should. For an active link connecting a node p with q, once p considers the link unreliable then q considers the link unreliable as well, possibly with a delay of one round. The *state of a node p at a round* consists of its name, the input value, and a set of its neighbors, with each incident link categorized as either passive, active, or unreliable, representing this categorization of links by the node p at the round. Links start as passive, except for one incident link per node initialized as active, then they may become active, and finally they may become unreliable.

A snapshot of the network at a node represents the node's knowledge of its connected component in the network restricted to the active edges and the states of its nodes. Formally, a *snapshot of network* at a node p at a round is a collection of states of some nodes that p has received and stores. A snapshot allows to create a map of a portion of the network, which is a graph with the names of nodes as vertices and the edges representing links. This map can include the input values of some nodes, should they become known. A connected component of a node with other nodes reachable by active links is a part of such a map. Formally, the *active connected component* of a node p at a round is a connected component, of the vertex representing p, in a graph that is a map of the network according to the snapshot of p at the round with only active links represented by edges.

algorithm OL-Agreement

```
1. initialize: Unreliable ← $\emptyset$, Active ← $\{\{p, q\}\}$ where $q$ is some neighbor,
   Passive ← set of links to $p$'s neighbors, except for the neighbor $q$ used in Active,
   state ← (name$_p$, input$_p$, Active, Passive, Unreliable),
   round ← 0, timestamp ← (state, round)
2. repeat
   (a) epoch ← round, Snapshot ← {state}
   (b) repeat
         i. round ← round + 1, add timestamp to set Timestamps
        ii. for each incident link $\alpha$ do
            A. if $\alpha$ is in Active then send Timestamps through $\alpha$
            B. if $\alpha$ is mature in Active and no message received through $\alpha$
                  then move $\alpha$ to Unreliable
            C. if a message received through $\alpha$ then place $\alpha$ in Active
       iii. for each received timestamp pair (state, $y$) do
            A. add (state, $y$) to Timestamps
            B. if $y >$ epoch then add state to Snapshot
   (c) until the active connected component is settled
   (d) if the active connected component is extendible then
         i. identify an outgoing edge as a connector
        ii. if the connector is incident to $p$ then place it in Active
3. until the active connected component is enclosed
4. set candidate$_p$ to the maximum input value in Snapshot
5. send pair (this-is-decision, candidate$_p$) through each active incident link
6. decide on candidate$_p$
```

Fig. 5. A pseudocode for a node p. In each round, node p checks to see if a pair of the form (decision, z) has been received, and if so then p forwards this pair through each active port, decides on z, and halts.

A node p sends a summary of its knowledge of the states of nodes in the network to the neighbors through all its active links at each round. If p receives a message with such knowledge from a neighbor, then p updates its knowledge and the snapshot by incorporating the newly learned information. At each round, a node p determines its active connected component based on the current snapshot. We say that *a node p has heard of a node q* if the name$_q$ occurs in the snapshot at p; the node p may either store some q's state or q's name may belong to a state of some other node that p stores. A node p considers another node q *settled* if p has q's state in its snapshot. A node p considers its active connected component *settled* if p has settled all the nodes in its active connected component. If a node p has heard about another node q such that q does not belong to the node p's active connected component, but it is connected to a node r in the active connected component by a passive link, then the node p considers the link connecting q to r as *outgoing*. If there is an outgoing link in p's active connected component then p considers its active connected component *extendible*, otherwise p considers its active connected component *enclosed*.

The algorithm is called OL-Agreement, its pseudocode is in Figure 5. Each node stores links it knows as unreliable in a set `Unreliable`, initialized to the empty set. Each node stores links it considers active in a set `Active`, initialized to some incident link. Each node stores passive links in a set `Passive`, which a node initializes to the set of all incident links except for the one initially activated link. All nodes maintain a variable `round` as a counter of rounds. In each round, a node creates a *timestamp pair*, which consists of its current state and the value of the round counter used as a timestamp. A node p stores timestamp pairs in a set `Timestamps`. For each node q different from p, a node p stores a timestamp pair for q if such a pair arrived in messages and only one pair with the largest timestamp. These variables are initialized by instruction (1) in Figure 5.

The initialization is followed by iterating a loop performed by instruction (2) in the pseudocode in Figure 5. The purpose of an iteration is to identify a new settled active connected component; we call an iteration *epoch*. An epoch is determined by the round in which it started, remembered in the variable `epoch` by instruction (2a). The knowledge of an active connected component of a node p identified in an epoch is stored in a set `Snapshot`, which is initialized at the outset of an epoch to the p's state by instruction (2a). This knowledge is represented as a collection of states of nodes that arrived to p in timestamp pairs, with timestamps indicating that they were created after the start of the current epoch, as verified by instruction (2(b)iiiB). The main part of an epoch is implemented as an inner repeat loop (2b). An iteration of this loop implements a round of communication with neighbors through active links and updating the state by instruction (2(b)ii).

An incident link in `Active` is *mature* if either it became active because a message arrived through it or p made it active spontaneously at some round i and the current round is at least $i + 2$. If a mature active link fails to deliver a message then p moves it to `Unreliable`. A set variable `Timestamps` stores timestamp pairs that a node sends in each message and updates after receiving messages at a round. A set variable `Snapshot` is used to construct an active connected component. `Snapshot` is rebuilt in each epoch, starting only with the current p's state. We separate storing timestamp pairs in a set `Timestamps` used for communication from storing states in `Snapshot` to build an active connected component, to facilitate a proper advancement of epochs in other nodes. We say that node p *completes the survey* of the network by a round if p has settled all the nodes in its active connected component according to the snapshot of this round. If the active connected component is extendible, then p identifies a *connector* which is an outgoing edge to be made active. We may identify an outgoing edge that is minimal with respect to the lexicographic order among all the outgoing links for a settled active connected component to be designated as a connector. If a connector is a link incident to p then p moves it to the set `Active`, by instruction (2d).

Theorem 6. *Algorithm* OL-Agreement *solves disconnected agreement in* $\mathcal{O}(nm)$ *rounds with fewer than* $2n$ *links used at any round and sending messages of* $\mathcal{O}(m \log n)$ *bits.*

Lower bounds for link usage. We now consider a setting in which the destinations of ports are not initially known to nodes. For any positive integers n and m such that $m = \mathcal{O}(n^2)$, we design a graph $\mathcal{G}(n, m)$ with $\Theta(n)$ vertices and $\Theta(m)$ edges, which makes any disconnected agreement solution to use $\Theta(m)$ links even if the nodes know the parameters n and m. We drop the parameters n and m from the notation $\mathcal{G}(n, m)$, whenever they are fixed and understood from context, and simply use $\mathcal{G}$. Consider any positive integers n and m such that $m = \mathcal{O}(n^2)$. Let graph $\mathcal{G}$ consist of two identical parts G_1 and G_2 as its subgraphs. The parts are $\lceil \frac{m}{n} \rceil$-regular graphs of $\lceil \frac{n}{2} \rceil$ vertices each. Without loss of generality, we can assume that the number $\lceil \frac{m}{n} \rceil$ is even, to guarantee that such regular graphs exist. Graph $\mathcal{G}$ is obtained by connecting G_1 and G_2 with $\lceil \frac{n}{2} \rceil$ edges such that each vertex from G_1 has exactly one neighbor in G_2. By the construction, graph $\mathcal{G}$ has $2 \lceil \frac{n}{2} \rceil = \Theta(n)$ vertices and $(\lceil \frac{m}{n} \rceil + 1) \lceil \frac{n}{2} \rceil = \Theta(m)$ edges. Let us assume now that the destinations of outgoing links are not initially known to the nodes. This means that ports can be associated with neighbors's names only after receiving messages through them. The following holds even if n, m can be a part of code.

Theorem 7. *For any disconnected agreement algorithm $\mathcal{A}$ relying on minimal knowledge and positive integer numbers n and m such that $n \leq m$ and $m \leq n^2$, there exists a network $\mathcal{G}(n, m)$ with $\Theta(n)$ nodes and $\Theta(m)$ links and an execution of algorithm $\mathcal{A}$ on $\mathcal{G}(n, m)$ that uses $\Theta(m)$ links.*

Theorem 8. *Let $\mathcal{A}$ be a disconnected agreement algorithm that uses $\mathcal{O}(n)$ reliable links concurrently when executed in networks with n nodes. For all natural numbers n and $\lambda \leq n$, there exists a network $\mathcal{G}$ with the stretch λ on which some execution of algorithm $\mathcal{A}$ takes $\Omega(n)$ rounds.*

Corollary 2. *If a disconnected agreement algorithm uses $\mathcal{O}(n)$ reliable links concurrently at any time, when executed in networks of n nodes, then this algorithm cannot be early stopping.*

8 Conclusion

We introduced the problem of disconnected agreement in the model of networks with links prone to failures such that faulty links may omit messages. This problem is of different nature than consensus or k-set agreement problems, which are typically considered in connected communication network, see the full version of the paper [8] for a related discussion. We measure the communication efficiency of algorithms by the size of individual messages or the number of non-faulty links used. This approach allows to demonstrate apparent trade-offs between running time and communication. One could study dependencies of the running time and the total number of messages exchanged or the total number of bits in messages sent by nodes executing disconnected-agreement algorithms. Another possible future direction of work concerns more severe link faults, for example such that result in delivering forged messages. Studying stretch of specific families of evolving networks is an open problem of independent interest.

References

1. Augustine, J., Pandurangan, G., Robinson, P., Upfal, E.: Towards robust and efficient computation in dynamic peer-to-peer networks. In: SODA 2012
2. Biely, M., Robinson, P., Schmid, U., Schwarz, M., Winkler, K.: Gracefully degrading consensus and k-set agreement in directed dynamic networks. Theoretical Computer Science **726**, 41–77 (2018)
3. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theoretical Computer Science **412**(40), 5602–5630 (2011)
4. Castañeda, A., Fraigniaud, P., Paz, A., Rajsbaum, S., Roy, M., Travers, C.: Synchronous t-resilient consensus in arbitrary graphs. In: Proceeding of the 21st Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS) (2019)
5. Charron-Bost, B., Függer, M., Nowak, T.: Approximate consensus in highly dynamic networks: The role of averaging algorithms. In: ICALP 2015 (2015)
6. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. Distributed Computing (2009)
7. Chlebus, B.S., Kowalski, D.R., Olkowski, J.: Fast agreement in networks with Byzantine nodes. In: Proceedings of the 34th International Symposium on Distributed Computing (DISC). LIPIcs, vol. 179, pp. 30:1–30:18 (2020)
8. Chlebus, B.S., Kowalski, D.R., Olkowski, J., Olkowski, J.: Consensus in networks prone to link failures. CoRR **abs/2102.01251** (2021)
9. Choudhury, A., Garimella, G., Patra, A., Ravi, D., Sarkar, P.: Crash-tolerant consensus in directed graph revisited (extended abstract). In: SIROCCO 2018 (2018)
10. Dolev, D.: The Byzantine generals strike again. Journal of Algorithms (1982)
11. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. Distributed Computing **1**(1), 26–39 (1986)
12. Hadzilacos, V.: Connectivity requirements for Byzantine agreement under restricted types of failures. Distributed Computing **2**(2), 95–103 (1987)
13. Khan, M.S., Naqvi, S.S., Vaidya, N.H.: Exact Byzantine consensus on undirected graphs under local broadcast model. In: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC). pp. 327–336. ACM (2019)
14. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: STOC (2010)
15. Kuhn, F., Moses, Y., Oshman, R.: Coordinated consensus in dynamic networks. In: Proceedings of the 30*th* Annual ACM Symposium on Principles of Distributed Computing (PODC 2011). pp. 1–10. ACM (2011)
16. Perry, K.J., Toueg, S.: Distributed agreement in the presence of processor and communication faults. IEEE Trans. on Software Engineering **12**(3), 477–482 (1986)
17. Santoro, N., Widmayer, P.: Time is not a healer. In: Proc., 6th Symp. on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 349, pp. 304–313 (1989)
18. Schmid, U., Weiss, B., Keidar, I.: Impossibility results and lower bounds for consensus under link failures. SIAM Journal on Computing **38**(5), 1912–1951 (2009)
19. Tseng, L.: Recent results on fault-tolerant consensus in message-passing networks. In: Proceedings of the 23rd Int. Colloquium on Structural Information and Communication Complexity (SIROCCO). LNCS, vol. 9988, pp. 92–108. Springer (2016)
20. Tseng, L., Vaidya, N.H.: Fault-tolerant consensus in directed graphs. In: Proc., ACM Symp. on Principles of Distributed Computing (PODC). pp. 451–460 (2015)
21. Tseng, L., Vaidya, N.H.: A note on fault-tolerant consensus in directed networks. SIGACT News **47**(3), 70–91 (2016)
22. Winkler, K., Schmid, U.: An overview of recent results for consensus in directed dynamic networks. Bulletin of EATCS **128** (2019)