Disconnected Agreement in Networks Prone to Link Failures

Bogdan S. Chlebus * Dariusz R. Kowalski * Jan Olkowski † Jędrzej Olkowski ‡

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 "disconnected agreement" the algorithmic problem of reaching such 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, size of messages, and running times of algorithms. A network has n nodes and m links. Nodes are normally assumed to know their own names and ability to associate communication with local ports. If we additionally assume that a bound Λ on stretch is known to all nodes, then there is an algorithm for disconnected agreement working in time $\mathcal{O}(\Lambda)$ using messages of $\mathcal{O}(\log n)$ bits. 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 failure, agreement, message size.

*School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia, USA.
†Department of Computer Science, University of Maryland, College Park, Maryland, USA.
‡Wydział Matematyki, Mechaniki i Informatyki, Uniwersytet Warszawski, Warszawa, Poland.

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 faulty link may not convey a message transmitted at a round.

Once a link omits a message, it may omit messages in the future. A link that has omitted a message manifested its faultiness and is considered *unreliable*. A safe approach for a communication algorithm is to treat unreliable links as if they crashed and not use them for transmissions after they manifested unreliability. Based on this intuition, we model a network with link failures as evolving through a chain of sub-networks obtained by removing unreliable links.

The disconnected agreement problem is about nodes reaching agreement on a common value. Each node starts with an input value and it eventually decides on some value. The requirements of reaching disconnected agreement are modeled after the problem of consensus. That problem is defined by the three requirements of termination, validity, and agreement, which constrain the output values and the process of deciding. Those conditions may depend on the nature of faults in a distributed system, for example there are different conditions specifying what the consensus problem is in the models of nodes prone to crashes and for arbitrary (Byzantine) faults of nodes. We address a problem like consensus in a model of dynamic networks whose topologies evolve because of removal of faulty links.

Deleting links affects the topology of a network, possibly even decomposing a network into disjoint connected components. If such a decomposition occurs fast enough, it may be impossible for some nodes to learn of input values in other components whose presence affects the ultimate decision for a given agreement-seeking algorithm. To make reaching agreement possible, we need either to restrict link failures, or to modify the specification of the agreement problem, or both.

In this work we study a scenario in which a faulty link may not transmit a message through at some rounds. Such an omission fault is a relatively benign type of link failure. Messages that are delivered, even through faulty links, are not tempered with and are received as transmitted. We do not impose any restrictions on which links may be faulty, in that, a removal of faulty links may produce an arbitrary sub-network. In particular, it is possible that some links simply crash such that this disconnects the network.

The problem to reach agreement needs to be suitably reformulated to reflect the possibility of a network becoming decomposed into disconnected sub-networks by removal of faulty 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. It is natural not to demand that nodes in different connected components decide on the same value, due to impossibility to accomplish such a goal without a proper flow of information. The process of breaking into connected components occurs in time, which creates a challenge for the nodes in a connected component to decide on the same value. Any two nodes may eventually end up in different connected components but what matters are connected components as they exist at the rounds in which such two nodes actually decide.

Our approach to specifying disconnected agreement subsumes the models of link crashes, which is stricter in that once a link manifests its faultiness by omitting a message it will omit all the messages in the future. Node crashes may also decompose a network into separate connected components, and our approach subsumes the model of node crashes as well. To see this for node crashes, observe that a node's crash could be simulated by crashing all the links incident to a node at the round of its crash. If a node's crash occurs in such a simulation then the node stays

operational but it constitutes a single-node connected component, so any decision by the node on some initial input value satisfies agreement and validity.

Let n denote the number of nodes and m the number of links in an initial network. Nodes are equipped with names, which are unique integers represented by $\mathcal{O}(\log n)$ bits. Initial input values are assumed to be encodable with $\mathcal{O}(\log n)$ bits. A message is considered "short" if it carries $\mathcal{O}(\log n)$ bits, and it is "linear" if it carries $\mathcal{O}(n \log n)$ bits. A network operates with "minimal knowledge" if each node initially knows its individual name, the initial input value, and it can distinguish among ports as sources of the incoming and outlets for the outgoing communication.

We use a network's dynamic attribute called "stretch," which is an integer determined by the number of connected components and their diameters. The purpose of using stretch is to consider scalability of disconnected agreement solutions to networks evolving through link failures and their removal from the network. In particular, we define "fast" and "early-stopping" algorithms solving disconnected agreement by referring to stretch. Along with time performance, we consider the amount of communication, measured as an upper bound on the number of bits in each individual message. We also study performance of algorithms measured by "link use" defined as the number of reliable links through which nodes transmit messages.

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. Faulty links may omit messages. 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 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 unknown 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 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 also show, in Section 7, that 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 developed in this paper, with their performance bounds, is given in Table 1.

The previous work on agreement in networks. Dolev [25] 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. [28], and Hadzilacos [31]. Khan et al. [35] 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 [52] presented necessary and sufficient conditions for the solvability of consensus in directed graphs under the models of

algorithm / section	time	message size	# links	knowledge	lower bound
FAST-AGREEMENT (Λ) / 3	Λ †	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$	Λ known	time $\lambda \leq \Lambda$
SM-AGREEMENT / 4	$n+1$	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$	minimal	time λ
LM-AGREEMENT / 5	$(\lambda+2)^3$	$\mathcal{O}(n \log n)$	$\mathcal{O}(m)$	minimal	time λ
ES-AGREEMENT / 6	$\lambda+2$ †	$\mathcal{O}(m \log n)$	$\mathcal{O}(m)$	minimal	time λ
OL-AGREEMENT / 7	$\mathcal{O}(nm)$	$\mathcal{O}(m \log n)$	$2n$ †	neighbors known	# links $\Omega(n)$

Table 1: A summary of the given deterministic distributed algorithms for disconnected agreement and their respective performance bounds. The notation n means the number of nodes and m the number of links in the initial network; none of these numbers is ever assumed to be known. The parameter Λ denotes an upper bound on stretch if it is known. The notation λ means the stretch actually occurring in an execution by the time all nodes halt. The dagger symbol † indicates the asymptotic optimality of the respective upper bound.

crash and Byzantine failures. In a related work, Choudhury et al. [21] provided a time-optimal algorithm to solve consensus in directed graphs with node crashes. Castañeda et al. [9] considered networks with nodes prone to crashes with an upper bound t on the number of crashes and showed that, as long at the network remained $(t+1)$-connected, Consensus was solvable in a number of rounds determined by how conducive the network was to broadcasting. Chlebus et al. [17] studied networks with Byzantine nodes such that the removal of faulty nodes leaves a network that is sufficiently connected; they gave fast solutions to consensus and showed a separation of consensus with Byzantine nodes from consensus with Byzantine nodes using message authentication, with respect to asymptotic time performance in suitably connected networks. Tseng [51] and Tseng and Vaidya [53] surveyed work on reaching agreement in networks subject to node faults. Winkler and Schmidt [55] gave a survey of recent work on consensus in directed dynamic networks.

The approach to failing links that we work with falls within the scope of modeling dynamic evolution of networks. Next, we discuss previous work on solving consensus in networks undergoing topology changes, malfunctioning links and transmission failures. Perry and Toueg [45], Santoro and Widmayer [47, 48], and Charron-Bost and Schiper [12] studied agreement problems in complete networks in the presence of dynamic transmission failures. Kuhn et al. [37] 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 [37] 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. Augustine et al. [4] studied dynamic networks with adversarial churn and gave randomized algorithm to reach almost-everywhere agreement with high probability in poly-logarithmic time. Charron-Bost et al. [11] considered approximate consensus in dynamic networks and provided connectivity restrictions on network evolution that make approximate consensus solvable. Coulouma et al. [24] characterized oblivious message adversaries for which consensus is solvable. Biely et al. [7] 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 [7] 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	time	message size	# links	knowledge	comments
Kuhn et. al. [36]	$O(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(m)$	minimal	for $T = \Omega(n)$
Biely et. al [7], alg. 2	$2D + 2E$	$O(m \log n)$	$\mathcal{O}(m)$	D known	directed links
Biely et. al [7], alg. 4	$3D + 3$	$\mathcal{O}(nD \log n)$	$\mathcal{O}(m)$	D known	k-set agreement
Kuhn et. al [37]	$O(n)$	$\mathcal{O}(m^2 \log n)$	$\mathcal{O}(m)$	minimal	Δ-coordinated

Table 2: A summary of the previous consensus results that are most relevant to the model studied in this paper and their respective performance bounds. Notation n denotes the number of nodes and m the number of links in an initial network; none of these numbers are ever assumed to be known. Letter T denotes a number such that in every interval of T rounds there exists a stable connected spanning subgraph, see [36]. Letter D denotes a dynamic source diameter, and E denotes a dynamic graph depth, see [7]. The stretch and parameters D and E in bidirectional networks with unreliable links are related as follows: $D = E = \Lambda \geq \lambda$. Parameter Δ is an upper bound on the difference in the respective times of termination for any two nodes, see [37].

algorithm includes D. Kuhn et al. [36] 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 [36] 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. Biely et al. [8] studied consensus in a synchronous model combining transient process and communication failures, with the numbers of such failures explicitly bounded, and demonstrated adaptability of popular consensus solutions to this model. Schmid et al. [50] showed impossibility results and lower bounds for the number of processes and rounds for synchronous agreement under transient link failures. Winkler et al. [56] investigated solving consensus in dynamic networks with links controlled by adversaries who only eventually provide a desired behavior of networks. A representative selection of previous results related to the model of this paper is given in Table 2.

Next, we briefly discuss advances on other algorithmic problems in dynamic networks. Cornejo et al. [23] considered the aggregation problem in dynamic networks, where the goal is to collect tokens, originally distributed among the nodes, at a minimum number of nodes. Haeupler and Kuhn [32] developed lower bounds on information dissemination in adversarial dynamic networks. Sarma et al. [49] investigated algorithms interpreted as random walks in dynamic networks. Michail et al. [42] studied naming and counting in anonymous dynamic networks. Michail et al. [43] studied propagation of influence in dynamic networks that may be disconnected at any round. Frameworks and models of distributed computing in networks evolving in time were discussed in [10, 36, 38, 41].

The previous work on the scalability of consensus solutions. We review the previous work on consensus solutions that scale well with their performance metrics, including communication and running time. Let the letter t denote an upper bound on the number of node failures that is known to all nodes, and the letter f be an actual number of failures occurring in an execution, which is not assumed to be known. We may call a consensus solution "fast" if it operates in time $\mathcal{O}(t + 1)$ and "early stopping" if its running time is $\mathcal{O}(f + 1)$. Scaling communication performance can be understood either "globally," which conservatively means $\mathcal{O}(n \text{ polylog } n)$ total communication, or "locally," which conservatively means $\mathcal{O}(\text{polylog } n)$ communication generated by each communicating agent.

Networks of degrees uniformly bounded by a constant provide an ultimate local scaling of communication. Upfal [54] showed that an almost-everywhere agreement can be solved in networks of constant degree with a linear number of crashes allowed to occur. Dolev et al. [27] gave an early stopping solution for consensus with arbitrary process failures and a lower bound $\min\{t+1, f+2\}$ on the number of rounds. Berman et al. [6] developed an early-stopping consensus solution with arbitrary process failures that was simultaneously optimal with respect to time performance $\min\{t+1, f+2\}$ and the number of processes $n > 3t$. Galil et al. [29] developed a crash-resilient consensus algorithm using $\mathcal{O}(n)$ messages, thus showing that this number of messages is optimal, but their algorithm runs in over-linear time $\mathcal{O}(n^{1+\varepsilon})$, for any $0 < \varepsilon < 1$; they also gave an early-stopping algorithm with $\mathcal{O}(n + fn^{\varepsilon})$ message complexity, for any $0 < \varepsilon < 1$. Chlebus et al. [18] presented a binary consensus algorithm operating in $\mathcal{O}(t + \log n)$ rounds and sending $\mathcal{O}(n + t \log t)$ messages for $t < \frac{n}{5}$; the algorithm sends the optimum number of bits/messages $\mathcal{O}(n)$ in the single-port model, as long as $t = \mathcal{O}(n/\log n)$. Garay and Moses [30] presented a fast consensus solution for arbitrary processor faults that runs on the optimum $n > 3t$ number of processors while using the amount of communication that is polynomial in n; they also gave an early-stopping variant of this algorithm. Chlebus and Kowalski [14] developed a gossiping algorithm coping with crashes and applied it to develop a consensus solution that is fast, by running in $\mathcal{O}(t+1)$ time, while sending $\mathcal{O}(n \log^2 t)$ messages, provided that $n - t = \Omega(n)$. Chlebus and Kowalski [15] developed a deterministic early-stopping algorithm that scales communication globally by sending $\mathcal{O}(n \log^5 n)$ messages. Coan [22] gave an early-stopping consensus solution that uses messages of size logarithmic in the range of input values; see also Bar-Noy et al. [5] and Berman et al. [6]. Chlebus at al. [19] gave a fast deterministic algorithm for consensus which has nodes send $\mathcal{O}(n \log^4 n)$ bits, and showed that no deterministic algorithm can be locally scalable with respect to message complexity. Chlebus and Kowalski [16] gave a randomized consensus solution terminating in the expected $\mathcal{O}(\log n)$ time, while the expected number of bits that each process sends and receives against oblivious adversaries is $\mathcal{O}(\log n)$, assuming that a bound t on the number of crashes is a constant fraction of the number n of nodes. Chlebus et al. [20] gave a scalable quantum algorithm to solve binary consensus, in a system of n crash-prone quantum processes, which works in $\mathcal{O}(\text{polylog } n)$ time sending $\mathcal{O}(n \text{ polylog } n)$ qubits against the adaptive adversary. Dolev and Lenzen [26] showed that any crash-resilient consensus algorithm deciding in $f+1$ rounds has worst-case message complexity $\Omega(n^2 f)$. Alistarh et al. [1] developed a randomized algorithm for asynchronous message passing that scales well to the number of crashes, in that it sends the expected $\mathcal{O}(nt + t^2 \log^2 t)$ messages.

A general perspective. Algorithmic problems concerning reaching agreement are central to the study of distributed systems and communication networks. The probem of consensus was introduced by Pease et al. [44] and Lamport et al. [39]. Books by Attiya and Welch [3], Herlihy and Shavit [34], Lynch [40], and Raynal [46] provide general expositions of formal models of distributed systems as frameworks to develop distributed algorithms in systems prone to failures of components. Attiya and Ellen [2] and Herlihy et al. [33] present techniques to show impossibilities and lower bounds for problems in distributed computing.

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. Nodes start with a read-only variable `name` initialized to their names; the private copy of this

variable at a node p is denoted by $\texttt{name}_p$. Every node identified other nodes by their names.

A node has distinctly labeled ports that act as interfaces to links connecting the node to its neighbors. We say that a node *knows neighbors* if it can associate with each port the name of a node that receives communication sent through this port and whose communication is received through this port. Links connecting pairs of nodes serve as bi-directional communication channels. Messages are scaled to channel capacity, in that precisely one message can be transmitted at a round through a link in each direction. The size of a message denotes the number of bits used to encode it. A message transmitted at a round is delivered within the same round. 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 called *final* for this execution and denoted by G_F.

Nodes start an execution of a distributed algorithm simultaneously. A node performes the following actions at a round: it sends messages through some of its ports, collects messages sent by neighbors, and performs local computation. A node can send messages to any subset of its neighbors and collect messages coming from all neighbors at a round.

Disconnected agreement. 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 seting 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.

An equivalent formulation of the agreement property is to say that the decisions of all the nodes in a connected component of G_F are equal. This is because if a node p decides at a round and q is a node that has already decided to which p is still connected by a reliable path at this round then if

that path stays reliable until halting then p and q end up in the same connected component of G_F.

The size of messages. 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*.

Stretch. Let H be a simple graph. If H is connected then $\mathrm{diam}(H)$ denotes the diameter of H. Suppose H has k connected components $C_1, \ldots, C_k$, where $k \geq 1$, and let $d_i = \mathrm{diam}(C_i)$ 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.

Knowledge. A property of distributed communication environments or executions is *known* if it can be used in codes of algorithms. 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. The number of nodes in a network n is never assumed to be known in this paper.

Neighbors can be discovered at one round of communication by all nodes sending their names to the neighbors: incoming messages allow to assign the sender's name to a port. This operation requires transmitting through every link in the network. If an algorithmic goal includes minimizing the number of used links then we assume that each node knows its neighbors prior to the beginning of an execution, in having the neighbors correctly mapped on ports.

The notation Λ will denote an upper bound on the stretch of a communication network in the course of an execution. If Λ is used then this means that Λ is known to all nodes.

Performance metrics. An initially connected graph G evolves through a nested sequence of its original sub-graphs, by removing edges representing faulty links. We want to assess how the running time an algorithm scales to the sub-networks resulting by removing faulty links. Stretch is used as a parameter capturing the challenge of solving disconnected agreement in sub-networks. The challenge in using stretch is that it evolves as a sequence of numbers. Let the notation λ_k mean a stretch of a network G at a round k of an execution, and λ be a stretch at the round of halting.

Proposition 1 *Stretches λ_k make a monotonously increasing sequence: $\lambda_i \leq \lambda_j$, for $i < j$.*

Proof: Consider an edge e belonging to a connected component C. If e gets removed from the graph and the connected component C stays intact then its diameter may only increase. Suppose the edge e is a bridge of C, and after its removal the connected component C breaks into two connected components C_1 and C_2. The stretch of the graph after removal of the edge e is the stretch before removal of this edge minus the diameter of C plus the diameters of C_1 and C_2, and this number incremented by 1. So this new stretch cannot be smaller than the original one, because the diameter of C is at most a sum of the diameters of C_1 and C_2 incremented by one. □

We consider bounds on performance of algorithms with respect to the stretch λ occurring at a round in which all nodes halt. Let $f : \mathbb{N} \to \mathbb{N}$ be a monotonously increasing function. For a communication algorithm to run in $\mathcal{O}(f(\lambda))$ time, it is sufficient and necessary that the running time is $\mathcal{O}(f(\lambda_k))$, for any round number k prior to halting, by Proposition 1.

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 known to all the nodes.

Our use of terms "fast" and "early stopping" follows a similar approach to consensus algorithms with node crashes. Let t denote an upper bound on the number of node crashes, which is known to all nodes, and f be an actual number of node crashes, which is not assumed to be known. Fast algorithms operate in running time $\mathcal{O}(t+1)$ and early stopping algorithms operate in running time $\mathcal{O}(f+1)$. We use Λ similarly as t and λ similarly as f.

3 Fast Agreement

We demonstrate the relevance of stretch to the running time of algorithms solving disconnected agreement. First, we show that a known upper bound on stretch allows to structure a disconnected agreement solution such that it operates in a number of rounds equal to the bound on stretch, so it is fast in our terminology. Second, we show that stretch is a meaningful yardstick to measure such running times, in that each algorithm solving disconnected algorithm may have to be executed for a number of rounds at least as large as the stretch.

If Λ is an upper bound on the stretch known to all nodes, then the following could be a possible approach to reach disconnected agreement in networks prone to link failures. Each node maintains a *candidate value* in a dedicated variable, which it initializes to the input. For Λ rounds, each node does two things per round: (1) sends the current candidate value to all neighbors, (2) updates the candidate value to the maximum of the current value and the values just received in messages. This algorithm is incorrect, which can be demonstrated on an example as follows. Let a network consist of three nodes: node p_1 with input 1, node p_2 with input 2, and node p_3 with input 3, all connected into a line $p_1 - p_2 - p_3$. Suppose the link $p_1 - p_2$ is non-faulty and $p_2 - p_3$ is faulty. The stretch is 2, as the stretch of a network starting a line of n nodes is always $n-1$. In the course of an execution, let the link $p_2 - p_3$ not deliver messages in the first round and deliver messages in the second round. The nodes p_1 and p_2 are in the same connected component $p_1 - p_2$, but p_1 decides on 2 while p_2 decides on 3, which violates the required agreement property.

We present next 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. Every node strives to eventually decide on the largest value possible, so a node remembers, in a variable `candidate`, only the largest value it has witnessed so far. This variable is initialized to the input value. At each round, a node either sends its `candidate` value to all neighbors or pauses in sending messages, then receives all incoming messages from the neighbors, if any, and updates `candidate` to the maximum value received in messages is some are greater than the current value. A node sends messages to neighbors at a round if either this is the first round, so `candidate` is the input value, or if `candidate` was updated to a new value in the previous round. We have that if a node sends a particular value to its neighbors then it does so only once. All nodes halt after Λ rounds, and every node decides on the `candidate` value, which is the largest value it has learned about.

In analyzing correctness of the algorithm, we asses when values of some magnitude carried in messages reach connected components of the network G_F. Let C be a connected component of graph G_F. We say that *value v reaches C* at a round if a node in C has its `candidate` value at least as large as v by this round. Graph $G_F - C$ denotes G_F with the nodes in C removed along with their incident edges.

`algorithm` FAST-AGREEMENT (Λ)

1. initialize `candidate` $\leftarrow$ `input`$_p$
2. `repeat` Λ `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`

Figure 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.

Lemma 1 *If a value reaches a connected component C of network G_F then this occurs by the round equal to the stretch of $G_F - C$ plus 1.*

Proof: The argument is by induction on the number of connected components of G_F. The base case occurs if G_F is connected. Then $G_F = C$, so $G_F - C$ is an empty graph of stretch 0. Any value that reaches G_F is at least as large as the input of some node, and inputs are available by round 1. Next comes the inductive step. Let C be a connected component of G_F, and assume that the inductive hypothesis holds for $G_F - C$. If a value v has reached C by the first round then we are done. Otherwise, a value at least as large as v has been brought in a message sent by a node p_1 to another node p_2, where p_1 is in $G_F - C$ and p_2 is in C. Let C' be the connected component of p_1 in $G_F - C$. By the inductive assumption, v reached C' by a round r equal to the stretch of $G_F - C - C'$ plus 1. After round r, the value traveled through C' for a number of rounds at most equal to the diameter of C', because a node transmits a value only in the round immediately following the round in which the value is received and becomes the new candidate value. After the journey through C', it takes one round to tranmit a value as large as v from p_1 to p_2, for a total number of rounds as large as the stretch of $G_F - C$ plus 1. This completes the inductive step, and so the argument by induction. □

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.*

Proof: The pseudocode in Figure 1 is structured as a loop of Λ iterations, each iteration taking one round. All nodes decide at the end, which gives the termination and time performance.

Each node p initializes its private variable `candidate`$_p$ to the input value `input`$_p$. An iteration of the repeat loop maintains an invariant that the variables `candidate` store only values taken from among the original input values, because only the values of this variable are sent and received. Every message carries only a candidate value. By the assumption on properties of input values, each such a value can be encoded with $\mathcal{O}(\log n)$ bits. Ultimately, each node p decides on a value in its `candidate`$_p$ variable, which gives validity.

Next, we show agreement. Let us consider a connected component C of G_F. Let v be the maximum value that has ever reached C. By Lemma 1, this value has reached C by the round

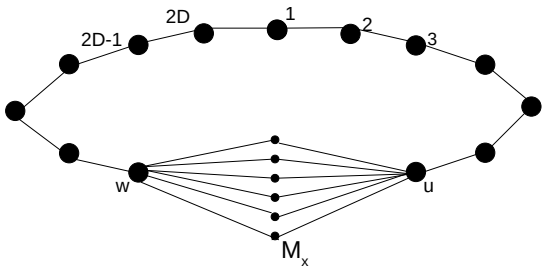

Figure 2: A depiction of graph $G(x, D)$ used in the proof of Lemma 2.

equal to the stretch of $G_F - C$ plus 1. Once v arrives at a node in C, this connected component C is flooded with value v, by design of the algorithm. This takes time equal to the diameter of C. It follows that flooding of C with v is completed by the round equal to the stretch of G_F. This is the round all nodes decide, so all the nodes in C decide on v. □

A lower bound on running time. We show a lower bound on the running time required to solve disconnected agreement in networks with link failures. This lower bound equals the stretch of the final graph of the network. We consider specific network topologies with the property that with no link failures the time needed to reach agreement is at least a diameter of the network. If a final graph is connected, then the stretch is the same as the diameter.

Lemma 2 *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 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.*

Proof: We define graphs $G(x, D)$, where $x > 0$ is an integer. Start with a cycle of length $2D$, and some three consecutive vertices u, v, and w on the cycle, where v is connected by edges with u and w. Replace vertex v by x copies of v, each connected precisely to the neighbors u and w of v. This construction is depicted in Figure 2, in which M_x denotes x copies of some vertex v. Observe that $G(x, D)$ has $2D + x - 1$ vertices and diameter D. Take $G = G(x, D)$ such that $n = 2D + x - 1$.

We consider executions of algorithm $\mathcal{A}$ on a network modeled by G in which no link ever fails. Each input value of a node in the network will be either 0 or 1. Start with the initial configuration C_0 in which all input values are 0, so decision is on 0 by validity. We proceed through a sequence of initial configurations $C_0, C_1, C_2, \ldots, C_n$, such that C_i has i nodes starting with the input value 1 and $n - i$ nodes starting with the input value 0. In particular, in configuration C_n all the nodes start with the input value 1, so decision has to be on 1, by validity. There exist two configurations C_i and C_{i+1} such that for C_i the decision is on 0 and for C_{i+1} the decision is on 1. These two configurations differ only in some vertex p having input 0 in C_i and input 1 in C_{i+1}.

Consider two executions of algorithm $\mathcal{A}$, one starting in configuration C_i and the other in C_{i+1}. Each node q of distance D from p sends and receives the same messages in the first $D - 1$ rounds in both executions. Take as q a node that is of distance D from p; such a node q exists by the specification of graph G. Node q needs to wait with deciding until at least the round D, because its state transitions in the first $D - 1$ rounds are the same in both executions, while the decisions are different. □

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.*

Proof: We use a network modeled by graph $G(x, D)$ used in the proof of Lemma 2, in which $x = 1$ and $n = 2D$. We have that $D = \frac{n}{2}$ rounds are necessary to reach agreement in some executions of algorithm $\mathcal{A}$, by Lemma 2. □

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

Proof: Let us take a network G with $n \geq 2\lambda$ nodes and with diameter λ such that some execution of algorithm $\mathcal{A}$ on G takes at least λ rounds with no link failures. Such a connected network exists by Lemma 2. The stretch of a connected network equals its diameter. □

Theorem 2 shows that algorithm Fast-Agreement(Λ) is asymptotically time-optimal on networks prone to link failures, because the actual stretch λ could be as large as an upper bound Λ on stretch.

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 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 following approach could possibly result in reaching disconnected agreement in networks prone to link failures, while sending only short messages. A node p maintains a set of input values in a dedicated variable, which it initializes to the singleton set with the node's input. The node p does two things per round: (1) for each neighbor, p sends some input value to this neighbor, but only such it has never sent before, (2) updates the set of known inputs value by adding new values just received in messages. A node halts after executing as many rounds as the size of the set of input values and decides on the maximum input value known. This algorithm works correctly if all the n input values are distinct, because each node collects all the input values in its connected component in the final graph. This algorithm is incorrect in general, which can be demonstrated on an example of a network of three nodes: node p_1 with input 1, node p_2 with input 1, and node p_3 with input 3, connected into a line $p_1 - p_2 - p_3$. To see incorrectness, suppose both links $p_1 - p_2$ and $p_2 - p_3$ are non-faulty. The node p_1 halts at the second round and decides on 1 and node p_3 halts at the third round and decides on 3, while all the nodes are in the same connected component.

Next, we give an algorithm for disconnected agreement that uses short messages while the nodes do not know the size n of the network nor any bound on stretch. Each node p has its name stored as $\texttt{name}_p$ and initial input value stored at $\texttt{input}_p$. A node p forms a pair $(\texttt{name}_p, \texttt{input}_p)$, which we call *input pair*, and sends it through each port as the first communication in an execution; simultaneously node p receives input pairs from its neighbors. Every other node q forms its respective pair

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 α do
   initialize set Channel$[\alpha]$ to empty ; send $(\texttt{name}_p, \texttt{input}_p)$ through α
3. for each port α do
   if a pair $(\texttt{name}_q, \texttt{input}_q)$ received through α 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 α do
   if some item in Inputs is not in Channel$[\alpha]$ then
   let x be the first such an item ; send x through α ; add x to Channel$[\alpha]$
   (b) for each port α do
   if a pair $(\texttt{name}_q, \texttt{input}_q)$ was just received through α 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

Figure 3: 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.

$(\texttt{name}_q, \texttt{input}_q)$. If $p \neq q$ then also $(\texttt{name}_p, \texttt{input}_p) \neq (\texttt{name}_q, \texttt{input}_q)$, even if $\texttt{input}_p = \texttt{input}_q$, so there are n input pairs in total. As an execution continues, nodes exchange input pairs among themselves. A message carries one input pair. The goal for each node is to learn as many input pairs as possible and help other nodes in accomplishing the same goal. A node halts in the first round whose number surpasses the number of input pairs the node has learned.

Next, we discuss how the algorithm is implemented. The algorithm is called SM-AGREEMENT, its pseudocode is in Figure 3. A node maintains a list Inputs of all input pairs that the node has ever learned about. For each port α, a node maintains a set Channel$[\alpha]$ storing all input pairs that have either been received through α or sent through α. At every round, a node verifies for each port α if there is a message to be sent through the port. This is determined by examining the list Inputs to check if there is an input pair in the list that does not belong to the set Channel$[\alpha]$: if this is the case then the first such a pair is transmitted through α. At every round, a node may receive a message through a port α: if this occurs, the node adds the received input pair to set Channel$[\alpha]$, unless the pair is already there, and appends the pair to the list Inputs, unless the pair is already in the list. A node maintains a counter of rounds called round, which is incremented in each round. An execution ends when this counter surpasses the size of the list Inputs.

We may consider a modification of algorithm SM-AGREEMENT, as presented in Figure 3, in which the repeat loop (4) is iterated indefinitely, rather than stopping after |Inputs| iterations. In an execution of the modified algorithm, eventually lists Inputs and sets Channel$[\alpha]$, for all ports α, stabilize in all nodes, so that messages are no longer sent. Suppose two nodes p and q are connected by a reliable path at a round when node q decides. By the round all the variables stabilize, the input pair $(\texttt{name}_r, \texttt{input}_r)$ will have reached node q, because there is a reliable path

from p to q. We need to show that this also occurs in an execution of algorithm SM-Agreement without the modification of the condition controlling the repeat loop, for the same link failures in the two executions. To this end, it suffices to demonstrate that node q withholds deciding long enough.

Lemma 3 *If nodes p and q are connected by a reliable path at a round when node q decides, and the list Inputs_p includes an input pair $(\mathit{name}_r, \mathit{input}_r)$ at that round, then the list Inputs_q includes this pair $(\mathit{name}_r, \mathit{input}_r)$ at the round when q decides.*

Proof: Let γ be a path between nodes r and p through which pair $(\mathtt{name}_r, \mathtt{input}_r)$ arrives to node p first; suppose $\gamma = (s_1, s_2, \ldots, s_k)$, where $r = s_1$ and $s_k = p$. Let δ be a reliable path connecting node p to q at a round when node q decides; suppose $\delta = (t_1, t_2, \ldots, t_\ell)$, where $p = t_1$ and $t_\ell = q$. We want to show that all input pairs originating at the nodes on the paths γ and δ reach node q by the round in which q decides. Consider a path ζ obtained by concatenating path γ with path δ at the node $p = s_k = t_1$. Let us denote the nodes on this path as $\zeta = (u_1, \ldots, u_\ell)$, where $u_1 = r$ and $u_\ell = q$, and denote by v_i a pair of the form $(\mathtt{name}, \mathtt{input})$ originating at u_i, for reference.

The simplest scenario, for node q to receive input pair $v_1 = (\mathtt{name}_r, \mathtt{input}_r)$, occurs when pair v_1 gets sent in each consecutive round along the path ζ until it reaches node q. During these transmissions, node q receives consecutive pairs v_i, starting from $v_\ell = (\mathtt{name}_q, \mathtt{input}_q)$ originating in q, through $v_1 = (\mathtt{name}_r, \mathtt{input}_r)$. In this scenario, a new pair gets added to list $\mathtt{Inputs}_q$ in each of these rounds, which makes node q postpone deciding for a total of at least ℓ rounds. We call this the *basic* scenario. Let us consider conceptual queues Q_i at each of the nodes u_i on ζ, for $i < \ell$, where the queue Q_i stores pairs that still need to be sent to u_{i+1}. These queues are manipulated by actions specified in instructions (4a) and (4b) in the pseudocode in Figure 3. In the basic scenario, each queue Q_i is initialized with u_i, and then each pair v_i moves through the queues Q_j, for $j = i+1, i+2, \ldots$ during consecutive rounds, to eventually arrive at $u_\ell = q$.

Next, we consider two possible alternative ways for input pairs to be sent along ζ to arrive at q, while departing from the basic scenario.

One possible departure from the basic scenario occurs when a pair v_i gets delivered to u_j, for $j > i$, by a shortcut outside of ζ rather than through consecutive nodes $u_{i+1}, u_{i+2}, \ldots, u_j$. This may result in pair v_i reaching q earlier than in the basic scenario, while postponing delivery of other pairs by one round. What does not change is that a new pair gets delivered to q at each round, so q keeps waiting, and also v_i gets added to Q_k to be later removed from Q_k exactly once, at each node u_k on the path ζ, for $k \geq i$, while sent down the path towards q, by the rules of manipulating lists `Inputs` and sets `Channel` specified in instructions (4a) and (4b) in the pseudocode in Figure 3. In this scenario, pair $(\mathtt{name}_r, \mathtt{input}_r)$ gets delivered to q no later than in the basic scenario.

Another possible departure occurs when a pair v originating at a node outside ζ gets delivered to a node on ζ and then travels along ζ towards q. Each such a pair v simply increases the number of pairs transmitted along ζ, so it may delay $v_1 = (\mathtt{name}_r, \mathtt{input}_r)$ from reaching q by one round. At the same time, such a pair v increments the size of the list `Inputs` at q by one when added to it, thereby extending the time period until q decides by one round. This makes node q wait long enough to receive input pair $(\mathtt{name}_r, \mathtt{input}_r)$ by the round it decides.

The two possible departures from the basic scenario discussed above can both occur in an execution, and also there could be multiple instances of pairs creating such scenarios departing from the basic one, without affecting the conclusion that node q waits long enough to receive pair $(\mathtt{name}_r, \mathtt{input}_r)$ prior to deciding. □

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

Proof: Input pairs stored in the list `Inputs` at a node have names of nodes as their first components, so there can be at most n pairs ever added to this list. A node decides immediately when a round number exceeds the size of list `Inputs`, which occurs by round $n+1$. This gives termination and the bound on running time. A node decides on an input value in an input pair present in its list `Inputs`, which stores pairs that originated at nodes of the network; this implies validity. By Lemma 3, all nodes in a connected component of the final graph G_F have identical lists `Inputs` at a round of deciding. A decision is on a maximum value in a list, which is uniquely determined by the set of pairs stored in the list; this gives agreement. A message carries one input pair, so it consists of $\mathcal{O}(\log n)$ bits, per the assumptions about the node names and input values. □

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 messages 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. We give an algorithm that works in $(\lambda+2)^3$ rounds, where λ is the stretch at the round of halting. 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.

A message of $\mathcal{O}(n \log n)$ bits allows to represent any set of node names. Once a node receives a message with names of nodes it can be certain that information about the magnitude of input values from these nodes has been received as well, assuming some relevant information about the input values was sent along. In principle, a node may halt as soon as it has heard from all the nodes in its connected component. The challenge is to detect such a moment, because a message of $\mathcal{O}(n \log n)$ bits can bring in many node names at once at a round, but still not all of them, and that round may be followed by a long stretch of rounds without receiving new names. To determine when to halt, we use timestamps, which are set to the current round number when created. A message of $\mathcal{O}(n \log n)$ bits can represent a set of pairs consisting of a node's name and a timestamp, as long as there is at most one such a pair for each node and the timestamp represents a round during an execution, while the running time of the algorithm is polynomial in n.

An overview of the algorithm. 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 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. If the node p at a round t_1 receives an epoch-serving timestamp pair $(\texttt{name}_q, t_2)$ for a node q that replaces a previously stored timestamp for q then the number $t_1 - t_2$ is called the *range of q at p*. Ranges are determined only for updated nodes at p, since q becomes updated after p receives $(\texttt{name}_q, t_2)$. A range of q is the length of a path traversed by a timestamp pair of q to reach p in the epoch, at the round the timestamp pair is received, so it is a lower bound on the distance from q to p 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.

An implementation of the algorithm. The algorithm is called LM-Agreement, its pseudocode is given in Figure 4. An execution starts with initialization of some variables by instruction (1). The main repeat loop follows as instruction (2); one iteration of the main repeat-loop makes an epoch. The pseudocode refers to a number of variables; we review them next.

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). We want the set `Nodes` to be calculated in two consecutive epochs at least once. To guarantee this, `PreviousNodes` is initialized to a special value denoted $\perp$ by instruction (2a) in the pseudocode, when the instruction is executed for the first time. The value $\perp$ is defined by the property that it is different from any set of nodes.

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 $\texttt{candidate}_p$

Figure 4: 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 correctness and running time. A node p is said to have *heard of node* q in an epoch, if the node p received a timestamp pair from q serving this epoch. We mean an epoch according to the node p, since epochs are not coordinated across the network, and so a different node q could be in a different epoch by its count of iterations of the main repeat loop (2) in the pseudocode in Figure 4.

Lemma 4 *If a node p has not heard of some node by a round of an epoch yet and that node is still connected to p by a reliable path at the next round, then the epoch continues through the next round.*

Proof: The proof is by induction on round numbers in an epoch. The node p starts an epoch with an empty set of timestamps pairs serving the epoch and adds at least its own timestamp pair to make itself updated at the first round of the epoch. If the node p has neighbors connected to it by reliable links, then, at the first round of the epoch, p hears of them and their status becomes updated as well. This means that an epoch never stabilizes at the first round, so it always continues beyond the first round. This provides the base step of induction.

Next we consider the inductive step. Let a round i of an epoch executed by the node p be such that node p has not heard of some node q in its current connected component up to round $i-1$, and

that such a node q stays connected to p during round i. If the node p has not heard of the node q by round $i-1$, then q's distance from p is greater than $i-1$ at the beginning of round i, so the distance is at least i. Let $\pi = (s_0, s_1, \ldots, s_\ell)$ be a shortest path from p to q, that exists at round i, where $p = s_0$ and $s_\ell = q$, for $\ell \geq i$. Node s_i has its timestamp pair delivered by s_1 to $p = s_0$ at round i, because such a timestamp pair has just completed its traversal of the path π towards p. This arrival establishes the range of s_i as i, because this is the distance to p at this round. If the node p has not heard of s_i before, then p changes the status of s_i from absent to updated. If the node p has heard of s_i before, then the range of s_i at p is at most $i-1$ after round $i-1$, so it gets changed to i. The epoch does not stabilize during round i in either case, so it continues through the next round $i+1$. □

Each epoch at a node p eventually comes to an end. This is because eventually all nodes in the connected component of p are either updated or stale at p, and their ranges stabilize to ones resulting from link failures that manifested themselves in the execution up to this point.

Lemma 5 *The duration of an epoch at every node is at most* $(\lambda+2)^2$.

Proof: Let us consider a node p. At a round i of an epoch, the status of every node of distance at most i from p at round i becomes either updated or stale at node p. Eventually a round k occurs such that the diameter of the connected component of p is k, and after this round node p will have heard of every node in its connected component. If a node q is of distance i from p at a round j, and this distance stays equal to i for at least i rounds following round j, then the range of every node on a shortest path of length i from p to q gets updated to its current value, which happens by round $j+i$. Let q be an arbitrary node. The distance from q to node p can change at most as many times as the diameter of the connected component of p at the round p ends the epoch. After each such a change, it takes up to as many rounds as the distance from p to q to have ranges of nodes on a shortest path from q to p updated to new values.

Let D denote the diameter of the connected component of node p when it ends the epoch. If $D = 0$ then the epoch ends after two rounds. If $D \geq 1$ then the round in which p ends the epoch is at most

$$D + \sum_{i=1}^{D-1} i^2 = D + \frac{D(D-1)}{2} \leq D^2 \ .$$

It follows that an epoch takes at most $(\lambda+2)^2$ rounds □

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

Proof: If a node p decides then the decision is on its candidate value stored in the variable `candidate`. Such a decision value is the initial input value of some node, by the instructions (1) and (2(b)iv) in Figure 4. This gives validity.

For agreement, we argue that when a node p decides, then it knows the maximum candidate value of all the nodes in its connected component at the round of deciding. The equality of sets `Nodes` and `PreviousNodes`, verified as condition (3) in the pseudocode in Figure 4 at the end of the epoch in which p decides, guarantees that the nodes of the connected components at the ends of the current and previous epochs have stayed the same. Consider the maximum candidate value present among the nodes of the connected component of p at the round of its deciding. This candidate value was also maximum among the values held by nodes of the connected component

of p in the previous epoch, since no nodes got disconnected from p in the current epoch. Suppose the maximum candidate value was held by a node q in the connected component at the end of the previous epoch. The node p hears from q in the current epoch, by Lemma 4. The candidate value of q travels along the same paths to p as the timestamp pairs from q. It follows that the node p decides on the candidate value of the node q, and this candidate value is maximum among candidate values of all nodes in the connected component at the round of deciding.

Next, we estimate the running time. A connected component of a node p can evolve through a sequence of contractions, occurring when the connected component shrinks and some nodes get disconnected to make other connected components. The number of epochs for every node is at least two, and it is at most the number of connected components of the final graph plus 1, for which the stretch plus 1 is an upper bound. An epoch at a node p takes at most $(\lambda + 2)^2$ time, by Lemma 5. The number of rounds by halting for a node is a sum of lengths of its epochs, so it is at most $(\lambda + 2)^3$.

Finally, we estimate the size of messages. A node sends its variable `Timestamps` and a candidate pair (this-is-candidate, `candidate`) in a message. The set `Timestamps` includes at most one timestamp pair per node. A node's name needs $\mathcal{O}(\log n)$ bits and a timestamp needs at most $\lg n^3 = \mathcal{O}(\log n)$ bits, because $\lambda < n$. Each candidate value is some original input of a node, so it also needs $\mathcal{O}(\log n)$ bits. □

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.

An overview of the algorithm. A graph whose vertices represent nodes and edges stand for links is like a map of the network at a round. Nodes executing the algorithm keep sending their knowledge of the network's map to neighbors, while simultaneously receiving similar information from them. The goal for each node is to build an approximation of a map of the network, which we call a snapshot. More precisely, a *snapshot* of the network at a node p consists of the names of nodes as vertices and edges between vertices representing links, but only these nodes and links that p has heard about. A snapshot at a node p may include the initial input for a vertex, should the node p know the input of a node represented by this vertex. A whole snapshot encoded as a message takes $\mathcal{O}(m \log n)$ bits. We explain how nodes manage snapshots next.

The model of minimal knowledge means that initially a node knows only its own name and input, but its ports are not labeled with names of the respective neighbors. To mitigate this, at the first round, every node sends its name through all the communication ports. At this very round, nodes receive the names of their respective neighbors coming through communication ports. This results in every node discovering its neighbors, which they use to assign neighbors' names to ports. During the first round, nodes do not send their input values. After the first round, a node has a snapshot consisting of its own name, the names of neighbors that submitted their names, and edges connecting the node to its newly discovered neighbors.

Starting from the second round, nodes iterate the following per round: send the current snapshot to each neighbor, receive snapshots from the neighbors, and update the snapshot by incorporating

`Algorithm` ES-AGREEMENT

1. initialize: $\texttt{Nodes} \leftarrow \{\texttt{name}_p\}$, $\texttt{Inputs} \leftarrow \{(\texttt{name}_p, \texttt{input}_p)\}$, $\texttt{Links} \leftarrow \emptyset$, $\texttt{Unreliable} \leftarrow \emptyset$
2. `for` each port `do`
   (a) send $\texttt{name}_p$ through this port
   (b) `if` $\texttt{name}_q$ received through this port `then`
      i. assign $\texttt{name}_q$ to the port as a name of the neighbor
      ii. add $\texttt{name}_q$ to `Nodes`; add edge $\{\texttt{name}_p, \texttt{name}_q\}$ to `Links`
3. `while` there exists 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 $\{\texttt{name}_p, \texttt{name}_q\}$ 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`

Figure 5: 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 $(\texttt{name}_q, ?)$ in $\texttt{Inputs}_p$.

newly acquired knowledge. Links may be marked as unreliable, if they have failed at least once to deliver a transmitted message. Edges representing unreliable links get removed from the snapshot. Such a removal is permanent, in that an edge representing an unreliable link is not restored to the snapshot. A snapshot at a node p determines a part of a connected component to which p belongs, but a snapshot may not be up to date, as links may fail and there could be a delay in p learning about it.

We say that *a node p has heard of a node q* if $\texttt{name}_q$ is a vertex in the snapshot at p. Each node hears of its neighbors by the end of the first round. A node q that belongs to the same connected component as a node p, according to the current snapshot at p, and such that p knows the initial input of q, is considered as *settled by p*.

A node p executing the algorithm participates in exchanging snapshots with neighbors, as long as its connected component contains nodes that have not been settled yet, according to the current snapshot. At the end of the first round, every node considers all its neighbors as pending settling. A soon as a node realizes that all nodes in its connected component according to the snapshot are settled, it sends its snapshot to the neighbors for the last time, finds the maximum input among the nodes in its snapshot, decides on this maximum value, and halts.

An implementation of the algorithm. The algorithm is called ES-AGREEMENT, its pseudocode is given in Figure 5. 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 `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 $(\texttt{name}_q, \texttt{input}_q)$ 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 5. 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 5. 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 5.

A node p has heard of a node q if $\texttt{name}_q$ is in the set $\texttt{Nodes}_p$. A node p has settled node q once the pair $(\texttt{name}_q, \texttt{input}_q)$ is in $\texttt{Inputs}_p$ and the node q belongs to the connected component of p according to its snapshot. We say that a *node p knows the state of a node q at the end of a round i* if the following inclusions hold: $\texttt{Nodes}_q \subseteq \texttt{Nodes}_p$, $\texttt{Links}_q \subseteq \texttt{Links}_p$, $\texttt{Unreliable}_q \subseteq \texttt{Unreliable}_p$, and $\texttt{Inputs}_q \subseteq \texttt{Inputs}_p$, where $\texttt{Nodes}_q$, $\texttt{Links}_q$, $\texttt{Unreliable}_q$, and $\texttt{Inputs}_q$ denote the values of these variables at q at the end of round i. If the node p hears of its neighbor q at the first round, then p knows only the q's name, but does not know either the input or any neighbor of q other than oneself.

The correctness and performance. We show that the algorithm is a correct disconnected agreement solution that is early stopping.

Lemma 6 *Once a node p settles a node q, then p knows the state of q at the end of the first round.*

Proof: After the first round, the set `Inputs` at q includes the only pair $(\texttt{name}_q, \texttt{input}_q)$. This is because of the initialization in instruction (1) of the pseudocode in Figure 5, and since q does not receive the neighbors' inputs at the first round, by instructions (2). Node q learns the names of its neighbors during the first round, which q uses to populate `Nodes` and `Links`. During the first iteration of the while loop in Figure 5, which occurs at the second round, the node q sends the pair $(\texttt{name}_q, \texttt{input}_q)$ to the neighbors, along with the contents of sets `Nodes`, `Links`, and `Unreliable`, as they were at the end of the first round. Pair $(\texttt{name}_q, \texttt{input}_q)$ spreads through the network carried in snapshots sent to neighbors, along with the contents of the sets `Nodes`, `Links`, and `Unreliable` at node q. When a pair $(\texttt{name}_q, \texttt{input}_q)$ reaches a node p for the first time, the contributions of the sets $\texttt{Inputs}_q$, $\texttt{Nodes}_q$, $\texttt{Links}_q$, and $\texttt{Unreliable}_q$ to the received snapshot are as from these set variables at the end of the first round at q. □

The nodes settle their neighbors by the end of the second round, after the first iteration of the while loop in Figure 5, as long as the links to these neighbors have not failed. As the while loop iterates in consecutive rounds, once a node p hears of some node q at a round i, then the earliest p is expected to settle the node q is at the next round $i+1$. To see this, observe that a node p hears of its neighbors at the first round and settles them in the second round, unless some links to neighbors failed. A failure of a link to a neighbor may postpone settling this neighbor, if its input value manages to reach p eventually, or p may never settle this neighbor, if it gets disconnected from p. The delay of at least one round between hearing of a node and settling this node is maintained through each iteration of the while loop. Once node p settles a node q, it learns the state of q at the end of the first round, by Lemma 6. Such a state may include the names of the nodes in $\texttt{Nodes}_q$ that are unsettled by p yet; if this is the case then the node p continues iterating the while loop.

After each round, a node builds a snapshot as an approximation of the network's topology. The vertices of this graph are the names of nodes from the set `Nodes`, and these links in the set `Links` that are not in `Unreliable` make the edges. We say that a node p *completes survey* of the network by a round if p has settled all nodes in its connected component according to the snapshot of this round. A node keeps communicating with neighbors until it completes survey, and then one more time, by instruction (4) in the pseudocode in Figure 5. This extra round of communication serves the purpose to help the neighbors complete their surveys in turn, as they may need the information that has just allowed the sender to complete its survey. Finally, a node decides on the maximum from the set of all the input values stored in the pairs in `Inputs`, and halts.

Lemma 7 *If a node p decides on $\texttt{input}_r$, another node q also decides, and the nodes p and q are connected by a reliable path at a round when they have already decided, then q has the node r as settled in its snapshot at this round.*

Proof: Suppose q does not have r in its snapshot as settled at the first round in which both p and q have completed surveys, to arrive at a contradiction. The node p has q in its connected component of the snapshot and, similarly, the node q has p in its connected component of the snapshot, because they are connected by a reliable path at the first round after completing surveys. Let $\gamma = (s_1, \ldots, s_k)$ be a reliable path connecting $p = s_1$ with $q = s_k$ at the first round in which both p and q have completed surveys and such that q settled p by a snapshot that arrived through this path. Let a node s_i on the path γ be such that i is the greatest index j of a node s_j in γ such that s_j settled r in its snapshot. An index i with this property exists because node $p = s_1$ is such. Moreover, the inequality $i < k$ holds because $q = s_k$ does not have r settled in its snapshot. There is a reliable path $\delta = (t_1, \ldots, t_m)$ from $r = t_1$ to $s_i = t_m$ through which a snapshot arrived first bringing $\texttt{input}_r$ to make s_i settle r. Consider a path ζ obtained by concatenating δ with a part of γ starting at s_i and ending at $s_k = q$, where $i < k$. Let us denote the nodes on this path as $(u_1, \ldots, u_\ell) = \zeta$, where $u_1 = r$ and $u_\ell = q$, for reference.

We examine the flow of information along ζ from $r = u_1$ towards $u_\ell = q$. At the first round, the node $u_\ell = q$ learns the name of its neighbor $u_{\ell-1}$. At the second round, node u_ℓ settles $u_{\ell-1}$ and learns of the node $u_{\ell-2}$, by Lemma 6. In general, a node u_j, such that $j > 1$, hears of its neighbor u_{j-1} at the first round and settles it at the second round. At a round a node u_j settles its neighbor u_{j-1}, it also hears of the node u_{j-2} as still unsettled. This creates a chain of dependencies such that the node u_j heard of a node up the path ζ towards r that is still unsettled and in the same connected component in its snapshot.

As snapshots with $\texttt{input}_r$ move along ζ towards q, this chain of dependencies, starting at a node that received $\texttt{input}_r$ most recently and ending at q, stays unbroken. This is because of the following two reasons. First, the part of ζ taken from δ provides reliable edges at all times, since $\texttt{input}_r$ manages to reach s_i: the only possibility of this not being the case would be to settle a node on this path via a different shorter path to s_i, but this is a shortest path by its choice. Second, the part consisting of γ provides reliable edges during the considered rounds, since these edges are still reliable when q completes survey. This makes node q eventually hear of r, and receive $\texttt{input}_r$ at the next round, which is a contradiction. □

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.*

Proof: A node decides on an input value from its snapshot, which gives validity.

We show agreement as follows. Consider two nodes p and q that are connected by a reliable path at the first round when each of these nodes has already decided. Suppose, to arrive at a contradiction, that node p decided on a value that is greater than the value that node q decided on. Let r be the node that provided its $\texttt{input}_r$ as the decision value for p. By Lemma 7, node q has node r settled at the round of deciding, so q decides on a value that is at least as large as $\texttt{input}_r$, which is a contradiction.

Next, we estimate the number of rounds needed for each node to halt. As an execution proceeds, information flows through the connected components of the network, by iteratively sending a snapshot to the neighbors and updating it at the same round. A value that gets decided on, in a particular connected component, may travel along a path that shares its parts with multiple connected components. If such a path crosses a connected component then its length is upper bounded by the connected component's diameter. If a link on such a path fails, this may occur after the future decision value got transmitted through this link, and the endpoints of this link may belong to different connected components. This shows that the number of hops a decision value makes on its way between a pair of nodes it at most the stretch. There are only two rounds that cannot be accounted for by this counting: the first round, during which the nodes discover their neighbors, and the last round, when a node notifies its neighbors of its snapshot for the last time. So if an execution terminates at a round t, then the stretch at this round is at least $t-2$. It follows that the algorithm terminates by the round $\lambda + 2$. □

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. Optimizing the link use in order to achieve an algorithmic goal could be interpreted as relying on a network backbone to accomplish the task and building such a backbone on the fly. 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 identify links, determined by the ports of a node, by the names of the respective neighbors of the node, and use the terms ports and incident links interchangeably. The disconnected agreement algorithm we describe next makes nodes halt in $\mathcal{O}(nm)$ rounds. 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. We also show that no algorithm can simultaneously be early stopping and use $\mathcal{O}(n)$ links.

An overview of the algorithm. The general idea of the algorithm is to have nodes build their maps of the network, representing the topology, that include the connected component of each node. In this the algorithm resembles ES-Agreement. 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. We want to accomplish manipulating snapshots by sending messages through as few links as possible, meaning $\mathcal{O}(n)$ links. This is possible in principle, because a spanning forest has $\mathcal{O}(n)$ links, and at the start each node already knows its neighbors. Input values could be a part of node attributes of the vertices on such a map. After the process of drawing a map is completed, which includes identifying a connected component, a decision can be made based on the information included in the 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. The states of a node may evolve in time, in that an incident link may change its categorization. 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.

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. Such new information may include either a state of a node q, such that p has never had a state of q, or a subsequent state of node q, such that p has already had some state of q. At each round, a node p determines its active connected component based on the current snapshot. If we refer to an active connected component of p then this means the active connected component according to the current snapshot. We say that *a node p has heard of a node q* if the $\texttt{name}_q$ 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*.

We want the nodes to participate in making some outgoing links active, each time the active connected component is settled and still extendible. All the nodes in the active connected component of a node p can choose the same outgoing link to make it active, once the active connected component becomes settled, because each node knows the same set of outgoing links. Once p's

`algorithm` OL-AGREEMENT

1. initialize: `Unreliable` $\leftarrow \emptyset$, `Active` $\leftarrow \{\{p, q\}\}$ where q is some neighbor,
   `Passive` $\leftarrow$ set of links to p's neighbors, except for the neighbor q used in `Active`,
   `state` $\leftarrow$ ($\texttt{name}_p, \texttt{input}_p, \texttt{Active}, \texttt{Passive}, \texttt{Unreliable}$),
   `round` $\leftarrow 0$, `timestamp` $\leftarrow$ (`state`, `round`)
2. `repeat`
   (a) `epoch` $\leftarrow$ `round`, `Snapshot` $\leftarrow \{\texttt{state}\}$
   (b) `repeat`
      i. `round` $\leftarrow$ `round` $+ 1$, add `timestamp` to set `Timestamps`
      ii. `for` each incident link α `do`
         A. `if` α is in `Active` `then` send `Timestamps` through α
         B. `if` α is mature in `Active` and no message received through α
            `then` move α to `Unreliable`
         C. `if` a message received through α `then` place α 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 $\texttt{candidate}_p$ to the maximum input value in `Snapshot`
5. send pair (this-is-decision, $\texttt{candidate}_p$) through each active incident link
6. decide on $\texttt{candidate}_p$

Figure 6: 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.

active connected component becomes settled and enclosed then p may decide.

An implementation of the algorithm. The algorithm is called OL-AGREEMENT, its pseudocode is in Figure 6. 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 6.

The initialization is followed by iterating a loop performed by instruction (2) in the pseudocode in Figure 6. 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. The inner repeat loop (2b) terminates once p completes the survey of the network, by condition (2c) controlling the loop. 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).

Once an epoch ends, by condition (2c), and the active connected component is enclosed, then the main repeat loop ends, by condition (3) controlling the loop. At this point, a node p is ready to decide, and the decision is on the maximum input value in a state stored in `Snapshot`, by instruction (4). Node p notifies each neighbor connected by an active link of the decision value, by instruction (5), and decides, by instruction (6). The pseudocode in Figure 6 omits what pertains to handling messages that could be generated in round (5) by some nodes. Namely, as a first thing at a round, a node p verifies if a pair of the form $(\texttt{decision}, z)$ has been received in the previous round, and if so then p forwards this pair through all active ports, decides on this value z, and halts.

The correctness and performance bounds. We show that algorithm OL-Agreement is a correct solution to disconnected agreement, and estimate its performance. The algorithm uses a similar paradigm to survey connected components as algorithm ES-Agreement. A node first learns of other nodes' names and later of these nodes' input values. This creates a chain of dependencies along paths through connected components, which in the case of algorithm OL-Agreement consist of active links.

Lemma 8 *If two nodes p and q are connected by a reliable path just after they both decided then each of them counts the other node in its last active connected component.*

Proof: Let us consider an arbitrary reliable path from p to q. Each link on this path is either active or passive. This is because once a passive link becomes active, then a message is sent trough it in each round, so a failure is immediately detected. A passive link is never activated only if it already

connects two nodes in the same active connected component. An active link makes its endpoints belong to the same active connected component. □

The following Lemma 9 is analogous to Lemma 7 about algorithm ES-Agreement, which also uses sufficiently large messages to build a map approximating the network. We need it to show agreement. A proof could be structured similarly to that of Lemma 7; we include a detailed argument for the sake of completeness.

Lemma 9 *If a node p decides on $\mathtt{input}_r$, another node q also decides, and nodes p and q are connected by a reliable path at a round when they have already decided, then q has node r as settled in its snapshot at this round.*

Proof: Suppose q does not have r in its snapshot as settled at the first round in which both p and q have completed their surveys, to arrive at a contradiction. The nodes p and q have each other in their active connected components, by Lemma 8. Let $\gamma = (s_1, \ldots, s_k)$ be a path consisting of the active links that connect $p = s_1$ with $q = s_k$ just after both p and q have completed their surveys and such that q settled p by a chain of `Timestamps` that arrived through this path. Let a node s_i on the path γ be such that i is the greatest index j of a node s_j in γ that has settled r in its snapshot. Such an index i exists because the node $p = s_1$ has this property. The inequality $i < k$ holds because $q = s_k$ does not have r settled in its snapshot. There is a path $\delta = (t_1, \ldots, t_m)$ from $r = t_1$ to $s_i = t_m$ consisting of active links through which `Timestamps` arrived first bringing $\mathtt{state}_r$ to enable s_i to settle r. Consider a path ζ consisting of the active links obtained by concatenating δ with a part of γ starting at s_i and ending at $s_k = q$, where $i < k$. Let us denote the nodes on this path as $(u_1, \ldots, u_\ell) = \zeta$, where $u_1 = r$ and $u_\ell = q$.

At the first round, the node $u_\ell = q$ learns a state of its neighbor $u_{\ell-1}$. At the second round, the node u_ℓ settles $u_{\ell-1}$ and hears of the node $u_{\ell-2}$. In general, a node u_j, such that $j > 1$, hears of its neighbor u_{j-1} at the first round and settles it at the second round. If a node u_j settles its neighbor u_{j-1} at a round then it also hears of the node u_{j-2} as still unsettled. This creates a chain of dependencies such that the node u_j heard of a node up the path ζ towards r that is still unsettled and in the same connected component in its snapshot.

As `Timestamps` with $\mathtt{state}_r$ move along ζ towards q, this chain of dependencies, starting at a node that received $\mathtt{state}_r$ most recently and ending at q, stays unbroken. This is because of the following two reasons. First, the part of ζ taken from δ provides active edges at all times, since $\mathtt{state}_r$ manages to reach s_i: the only possibility of this not being the case would be to settle a node on this path via a different shorter path to s_i, but this is a shortest path by its choice. Second, the part consisting of γ provides active edges during the considered rounds, since these edges are still reliable when q completes the survey. This makes node q eventually hear of r, and receive $\mathtt{state}_r$ at the next round, which is a contradiction. □

We consider an auxiliary *activation process* on a graph that models the activation of connectors in an execution of algorithm OL-Agreement. The process acts on a given simple graph H that has some k vertices and proceeds through consecutive rounds. An edge of H may progress through three states, first passive, then possibly active, and then possibly be deleted. We consider subgraphs determined by active edges, so connected components are active connected components. An edge connecting a vertex in a connected component C to a vertex in a different connected component is considered as outgoing from C. In the beginning of a round, if an active connected component has an outgoing edge, then one such an edge is made active. At the end of a round, some active edges may be deleted. The process continues until there are no outgoing edges.

Lemma 10 *In the course of the activation process on a graph with k vertices, the number of active edges in the graph at any round is at most $2k - 2$.*

Proof: We start with no active edges, so each vertex is an active connected component and k is the number of such components. Consider the activation of new edges at a round as occurring sequentially. As an edge is activated, it may connect two different active connected components, thus decreasing their number, or it may close a cycle. Suppose an active connected component C_1 gets connected to an active connected component C_2, then C_2 to C_3, and so on through C_i, with the edge activated in C_i connecting C_i to some C_j, for $1 \leq j < i$. That last edge from a vertex in C_i to a vertex in C_j is not needed to make all C_i, for $1 \leq j \leq i$, into one connected component, so we treat it as an *extra* edge. A number of such extra edges created at a round is maximized when $i = 2$, because for each decrease of the number of connected components with one activated edge we also activate another edge. A tree minimizes the number of connected components to one, and a tree on k vertices has $k - 1$ edges. We obtain that the number of active edges at each round is at most twice the number of edges in a tree of k vertices, which is $2(k - 1)$. □

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.*

Proof: A node decides on an input value from a state in its snapshot. All states include original input values, which gives validity.

Consider two nodes p and q connected by a reliable path at the first round when each of these nodes has already decided. We want to show that p and q decide on the same value. Suppose, to arrive at a contradiction, that the node p has decided on a value that is greater than the value that the node q has decided on. Let r be the node whose state provided its input value as the decision value for p. By Lemma 9, the node q has the node r settled at the round of deciding, so q decides on a value that is at least as large as $\texttt{input}_r$, which is a contradiction. This gives agreement.

As an execution proceeds, information flows through each active connected component, by iteratively sending timestamp pairs to neighbors via active links and simultaneously updating the latest states in timestamp pairs. The length of a path of active links traversed by a timestamp pair may be as long as the number of nodes in an active connected component minus one. This means that an epoch takes fewer than n rounds. After a new connector is added, it takes the length of an epoch for all the nodes to settle on the states of the nodes in an active connected component. There may be up to m links added as connectors. This means that every node halts in $\mathcal{O}(nm)$ rounds.

The process of making links active could be modeled as the activation process on a graph representing the network. By Lemma 10, the number of links that are active at the same time is less than $2n$. □

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 Theorem 7 holds even if n and 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.*

Proof: If a node p executing $\mathcal{A}$ sends a message by an incident link, through which it has not received nor sent any messages yet, then the receiving node could be any original neighbor of the node p in the network $\mathcal{G} = \mathcal{G}(n,m)$ which has not communicated with p yet. We refer as *uncovered neighbors* of p at a round to these among p's neighbors that p has not received a message from nor sent a direct message to yet. Choosing a node that is an uncovered p's neighbor, identifiable only by its port at p, is a part of the adversarial strategy of failing links.

Algorithm $\mathcal{A}$ gets executed on some initial configurations of the network $\mathcal{G}$, as defined above. Let the notation $\mathcal{C}_{i,j}$ mean an initial configuration in which the nodes from the subgraph G_1 have $i \in \{0,1\}$ as their input values while the nodes from the subgraph G_2 have $j \in \{0,1\}$. We call a link between two nodes in G_1 *unused*, by a given round, if none of its endpoints has tried to send a message through it yet. Let k_1 denote the number of nodes in G_1 that have at least one unused link incident to some other node in G_2. At the beginning, $k_1 = |G_1|$ and k_1 may decrease in the course of an execution.

For any initial configuration $\mathcal{C}_{i,j}$, where $i,j \in \{0,1\}$, consider the following adversarial strategy to fail links. Let a node p be in G_1 and let ℓ denote a link that p wants to send a message through at a round. Suppose node p has at least two unused links but wants to send a message by only one of them. The adversary allows the message to be delivered and the other endpoint of ℓ is chosen to be an arbitrary uncovered neighbor of p in G_1. Suppose there is only one unused link incident to p and the node p wants to send a message through it. If $k_1 > 1$ then let this link ℓ fail before it delivers a message, and otherwise, if $k_1 = 1$, then let link ℓ deliver the message to a remaining unassigned neighbor of p. It follows that in this case the message must go outside G_1. Each time number k_1 decreases by one, then all but one reliable links incident to some node of G_1 have been used and they will not be failed by the adversary in the continuation of the execution. Hence, before the first message is delivered from some node in part G_1 to some node in part G_2, at least $(\lceil \frac{m}{n} \rceil + 1) \cdot (|G_1| - 1)$ reliable links will have been used for communication between the nodes in G_1. A similar reasoning applies to communication within part G_2 of $\mathcal{G}$.

Assume now, to arrive at a contradiction, that in all executions starting from the configurations $\mathcal{C}_{0,0}$, $\mathcal{C}_{0,1}$, and $\mathcal{C}_{1,1}$, in which the above strategy has been used, fewer than $(\lceil \frac{m}{n} \rceil + 1) \cdot (|G_1| - 1)$ non-faulty links have been used. We call these executions $\mathcal{E}_0$, $\mathcal{E}_b$, and $\mathcal{E}_1$, respectively. Nodes in part G_1 do not communicate with nodes in G_2 in any of these executions. For nodes in the part G_1, an execution starting from $\mathcal{C}_{0,0}$ is indistinguishable from an execution starting from $\mathcal{C}_{0,1}$. Similarly, for nodes in the part G_2, an executions starting from $\mathcal{C}_{0,1}$ is indistinguishable from an execution starting from $\mathcal{C}_{1,1}$. It follows that each node in the part G_1 decides on the same value in the execution $\mathcal{E}_b$ as in the execution $\mathcal{E}_0$, and a decision has to be on 0 by validity. Each node in the part G_2 decides on the same value as in the execution $\mathcal{E}_1$, and a decision is on 1, by validity. This is a contradiction with agreement, as both parts G_1 and G_2 are connected in $\mathcal{E}_b$ and have to decide

on the same value. Therefore, in one of the considered executions, more than these many reliable links have to be used: $(\lceil \frac{m}{n} \rceil + 1) \cdot (|G_1| - 1) = (\lceil \frac{m}{n} \rceil + 1) \cdot (|G_2| - 1)$, which is $\Theta(m)$. □

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.*

Proof: Let n and $\lambda \leq n$ be natural numbers; we assume that both n and λ are even to simplify the notations. Let H_1 be a network with $\frac{n}{2}$ nodes and diameter $\frac{\lambda}{2} - 1$. Let H_2 be a copy of H_1. Let us form a network $\mathcal{G}$ by taking the nodes in H_1 and H_2 and adding all possible links between the nodes in H_1 and H_2. The diameter of $\mathcal{G}$ is at most $2 \cdot (\frac{\lambda}{2} - 1) + 1 = \lambda - 1$.

The following is an adversarial strategy to fail links. For a round i, let K_i be a set of all the reliable links between H_1 and H_2 by which nodes attempt to send messages at this round. The adversary fails all the links from K_i before any message arrives, as long as H_1 and H_2 stay connected. Because there are $\frac{n^2}{4}$ links between the nodes in parts H_1 and H_2, this guarantees that no two nodes, of which one is in H_1 and the other in H_2, exchange a message during at least $\Omega(\frac{n^2}{4n}) = \Omega(n)$ rounds.

Let us consider the following three initial configurations of G. In the first configuration $\mathcal{I}_1$, all the nodes start with input values 0. In the second configuration $\mathcal{I}_2$, all the nodes start with inputs 1. In the third configuration $\mathcal{I}_3$, the nodes from H_1 start with the input 0, while the nodes from H_2 start with the input 1. We consider executions of algorithm $\mathcal{A}$ starting with each of the initial configurations $\mathcal{I}_k$, for $k = 1, 2, 3$, when the adversary applies the strategy described above to fail edges; let $\mathcal{E}_k$ be the respective execution. If all the nodes halt before some pair of nodes such that one is from H_1 and the other is from H_2 communicate among themselves, then the nodes in H_1 cannot distinguish the execution $\mathcal{E}_1$ from $\mathcal{E}_3$ and the nodes in H_2 cannot distinguish the execution $\mathcal{E}_2$ from $\mathcal{E}_3$. Let us consider decisions by the nodes in execution $\mathcal{E}_3$. The nodes in H_1 decide on 0, as they should in $\mathcal{E}_1$, by validity, while the nodes in H_2 decide on 1, as they should in $\mathcal{E}_2$, by validity. This contradicts the requirement of agreement. □

We conclude by settling the question if an algorithm can be simultaneously early stopping and optimize link use.

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.*

Proof: Let us consider integers n and $\lambda_n \leq n$ such that $\lambda_n = o(n)$. By Theorem 8, the algorithm works in $\Omega(n)$ rounds. The algorithm cannot be early stopping, because this would mean running in time $\mathcal{O}(\lambda_n) = o(n)$. □

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. Disconnected agreement is a variant of consensus that has the agreement condition reformulated such that nodes have to agree on the same value only if they belong to the same connected component of a network obtained by removing the faulty links. The number of values that nodes decide on can be as large as the number of connected components. As far as allowing for different decision values, the problem is similar to k-set agreement, in which

agreement is relaxed such that nodes may agree on at most k values, for a positive integer k. The essential difference between the two problems is that an acceptable number of decision values in k-set agreement is known in advance while the disconnected agreement has the admissible number of decision values enforced by the changes of topology, and in particular if the network stays connected then all the nodes have to decide on the same value. An interesting aspect of k-set agreement, as compared to consensus, is that it can be solved in time $\lfloor \frac{t}{k} \rfloor + 1$ with up to t node crashes in synchronous message-passing systems, and this time is necessary, see [13], while consensus requires $t + 1$ rounds. The motivation for disconnected agreement is not to relax consensus such that it can be solved faster, but rather to investigate what running time and size of messages is needed to solve a problem defined by (1) not imposing any restrictions on which links in a network fail and omit messages, and (2) minimally relaxing the agreement condition such that solutions exist. We showed such trade-offs between running time and size of messages, and also looked into minimizing the number of used links. A possible future direction of work concerns more severe link faults, for example such that result in delivering forged messages.

We demonstrated that a dynamic stretch of a network is a meaningful parameter that serves as a yardstick to measure running time efficiency of disconnected agreement algorithms. The most running-time efficient disconnected-agreement algorithms are early stopping in that they work in $\mathcal{O}(\lambda)$ rounds, where λ is the stretch occurring in an execution. We showed how to design such an algorithm that uses messages of $\mathcal{O}(m \log n)$ bits. Trading off running time for size of messages, we developed an algorithm using smaller messages of only $\mathcal{O}(n \log n)$ bits that runs in $(\lambda+2)^3$ rounds. The running time $(\lambda + 2)^3$ of this disconnected-agreement algorithm could possibly be improved asymptotically, while still using messages of $\mathcal{O}(n \log n)$ bits, but we are sceptical if it is possible to design an early stopping disconnected-agreement algorithm that uses messages of a size that is significantly smaller than $\Theta(m \log n)$ bits.

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.

References

[1] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. *Distributed Computing*, 31(6):489–501, 2018.

[2] Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.

[3] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, Second edition, 2004.

[4] John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 551–569. SIAM, 2012.

[5] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, 1992.

[6] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 1992.

[7] Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theoretical Computer Science*, 726:41–77, 2018.

[8] Martin Biely, Ulrich Schmid, and Bettina Weiss. Synchronous consensus under hybrid process and link failures. *Theoretical Computer Science*, 412(40):5602–5630, 2011.

[9] Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. Synchronous t-resilient consensus in arbitrary graphs. In *Proceeding of the 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 11914 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2019.

[10] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[11] Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015), Part II*, volume 9135, pages 528–539. Springer, 2015.

[12] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[13] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, 2000.

[14] Bogdan S. Chlebus and Dariusz R. Kowalski. Robust gossiping with an application to consensus. *Journal of Computer System and Sciences*, 72(8):1262–1281, 2006.

[15] Bogdan S. Chlebus and Dariusz R. Kowalski. Time and communication efficient consensus for crash failures. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2006.

[16] Bogdan S. Chlebus and Dariusz R. Kowalski. Locally scalable randomized consensus for synchronous crash failures. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 290–299. ACM, 2009.

[17] Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski. Fast agreement in networks with Byzantine nodes. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[18] Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski. Deterministic fault-tolerant distributed computing in linear time and communication. In *Proceedings of the 42nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 344–354. ACM, 2023.

[19] Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 111–120. ACM, 2009.

[20] Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Scalable quantum consensus for crash failures. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[21] Ashish Choudhury, Gayathri Garimella, Arpita Patra, Divya Ravi, and Pratik Sarkar. Crash-tolerant consensus in directed graph revisited (extended abstract). In *Revised Selected Papers of the 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 11085 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2018.

[22] Brian A. Coan. Efficient agreement using fault diagnosis. *Distributed Computing*, 7(2):87–98, 1993.

[23] Alejandro Cornejo, Seth Gilbert, and Calvin C. Newport. Aggregation in dynamic networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 195–204. ACM, 2012.

[24] Étienne Coulouma, Emmanuel Godard, and Joseph G. Peters. A characterization of oblivious message adversaries for which consensus is solvable. *Theoretical Computer Science*, 584:80–90, 2015.

[25] Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[26] Danny Dolev and Christoph Lenzen. Early-deciding consensus is expensive. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279. ACM, 2013.

[27] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

[28] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

[29] Zvi Galil, Alain J. Mayer, and Moti Yung. Resolving message complexity of Byzantine agreement and beyond. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 724–733. IEEE, 1995.

[30] Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

[31] Vassos Hadzilacos. Connectivity requirements for Byzantine agreement under restricted types of failures. *Distributed Computing*, 2(2):95–103, 1987.

[32] Bernhard Haeupler and Fabian Kuhn. Lower bounds on information dissemination in dynamic networks. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, volume 7611 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2012.

[33] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[34] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[35] Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact Byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 327–336. ACM, 2019.

[36] Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the* 42*nd ACM Symposium on Theory of Computing (STOC)*, pages 513–522. ACM, 2010.

[37] Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *Proceedings of the* 30*th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–10. ACM, 2011.

[38] Fabian Kuhn and Rotem Oshman. Dynamic networks: models and algorithms. *SIGACT News*, 42(1):82–96, 2011.

[39] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[40] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[41] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[42] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *Proceedings of the* 15*th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8255 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2013.

[43] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Causality, influence, and computation in possibly disconnected synchronous dynamic networks. *Journal of Parallel and Distributed Computing*, 74(1):2016–2026, 2014.

[44] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[45] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, 1986.

[46] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[47] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proceedings of the* 6*th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 349 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 1989.

[48] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3):232–249, 2007.

[49] Atish Das Sarma, Anisur Rahaman Molla, and Gopal Pandurangan. Fast distributed computation in dynamic networks via random walks. In *Proceedings of the* 26*th International Symposium on Distributed Computing (DISC)*, volume 7611 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2012.

[50] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[51] Lewis Tseng. Recent results on fault-tolerant consensus in message-passing networks. In *Proceedings of the* 23*rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 9988 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2016.

[52] Lewis Tseng and Nitin H. Vaidya. Fault-tolerant consensus in directed graphs. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460. ACM, 2015.

[53] Lewis Tseng and Nitin H. Vaidya. A note on fault-tolerant consensus in directed networks. *SIGACT News*, 47(3):70–91, 2016.

[54] Eli Upfal. Tolerating a linear number of faults in networks of bounded degree. *Information and Computation*, 115(2):312–320, 1994.

[55] Kyrill Winkler and Ulrich Schmid. An overview of recent results for consensus in directed dynamic networks. *Bulletin of EATCS*, 128, 2019.

[56] Kyrill Winkler, Manfred Schwarz, and Ulrich Schmid. Consensus in rooted dynamic networks with short-lived stability. *Distributed Computing*, 32(5):443–458, 2019.