# Making Asynchronous Distributed Computations Robust to Channel Noise*†

**Keren Censor-Hillel[1], Ran Gelles[2], and Bernhard Haeupler[3]**

1 **Department of Computer Science, Technion, Israel**
ckeren@cs.technion.ac.il
2 **Faculty of Engineering, Bar-Ilan University, Israel**
ran.gelles@biu.ac.il
3 **Department of Computer Science, Carnegie Mellon University, USA**
haeupler@cs.cmu.edu

## Abstract

We consider the problem of making distributed computations robust to noise, in particular to worst-case (adversarial) corruptions of messages. We give a general distributed interactive coding scheme which simulates any asynchronous distributed protocol while tolerating a maximal corruption level of $\Theta(1/n)$-fraction of all messages. Our noise tolerance is optimal and is obtained with only a moderate overhead in the number of messages.

Our result is the first *fully distributed* interactive coding scheme in which the topology of the communication network is not known in advance. Prior work required either a coordinating node to be connected to all other nodes in the network or assumed a synchronous network in which all nodes already know the complete topology of the network.

Overcoming this more realistic setting of an unknown topology leads to intriguing distributed problems, in which nodes try to learn sufficient information about the network topology in order to perform efficient coding and routing operations for coping with the noise. What makes these problems hard is that these topology exploration computations themselves must already be robust to noise.

**1998 ACM Subject Classification** C.2.4 Distributed Systems, E.4 Coding and Information Theory

**Keywords and phrases** Distributed Computation, Coding for Interactive Communication, Noise-Resilient Computation, Coding Theory

**Digital Object Identifier** 10.4230/LIPIcs.ITCS.2018.50

## 1 Introduction

Fault tolerance is one of the central challenges in the design of distributed algorithms. Typically, computation is performed by $n$ nodes, of which some subset may be *faulty* and not behave as expected. This includes *crash* or *Byzantine* failures. Faults can also occur as communication errors, if links suffer from, e.g., *omissions*, *alterations* or *Byzantine* errors (see, e.g., [31, 2]).

We focus on alteration errors, in which the content of sent messages may be corrupted. Previous work in the setting of faulty channels provides fault-tolerant algorithms either for

* Research supported in part by the Israel Science Foundation (grants 1696/14 and 1078/17), the Binational Science Foundation (grant 2015803), and NSF grants CCF-1527110 and CCF-1618280.
† A full version of this paper is available at [11], https://arxiv.org/abs/1702.07403.


specific tasks such as the leader election or the consensus problem (e.g., [36, 39, 24, 40]), or for a specific class of topologies (e.g., [32]).

In this paper, we provide a general technique that takes as an input an asynchronous distributed protocol over an arbitrary topology and outputs a simulation of this protocol that is resilient to noise. Specifically, we develop several tools whose combination allows us to obtain the first *fully distributed* interactive coding scheme.

**The Challenge.** In order to tolerate channel noise and preserve the correctness of computations, sophisticated coding techniques must be employed. Several works (e.g., [35, 28, 27, 1]) provide such coding schemes, however, they all assume that the network's topology is known in advance. We wish to challenge this assumption, and allow truly distributed coding schemes.

Interestingly, once communication is unreliable, even the simplest distributed tasks, such as flooding information over the network or constructing a BFS tree, become tremendously difficult to execute correctly. For instance, the asynchronous distributed Dijkstra or Bellman-Ford algorithms [33] miserably fail when messages may be corrupted. To see why, recall that in the Bellman-Ford algorithm, each node sends to all of its neighbors its distance from the root. A node then sets its neighbor that is closest to the root as its parent. However, if messages are incorrect, the distance mechanism may fail and nodes may set their parents in an arbitrary way.

**Our Contribution.** In any attempt to tolerate message corruptions, naturally, some bound on the noise must be given. Indeed, if a majority of the sent messages are corrupted, there is no hope to complete a computation correctly. On the other hand, when the noise falls below a certain threshold, fault tolerant computation can be obtained, for example, by employing various coding techniques.

The field of *coding for interactive communication* (see, e.g., the survey of [19]) considers the case where two or more parties carry some computation by sending messages to one another over noisy channels and strives to devise *coding schemes* with good guarantees. A coding scheme is a method that is given as an input a protocol $\pi$ that assumes reliable channels, and outputs a noise-resilient protocol $\Pi$ that simulates the communication of $\pi$. The two main measures upon which a coding scheme is evaluated are its *noise resilience* – the fraction of noise that the resilient simulation $\Pi$ can withstand – and its *overhead* – the amount of redundancy $\Pi$ adds in order to tolerate faults. For networks with $n$ nodes, it is easy to show that the maximal adversarial noise fraction that any resilient protocol can cope with is $\Theta(1/n)$ [28]. Indeed, if more than $(1/n)$-fraction of the messages are corrupted, then the noise can completely corrupt all the communication of the node that sends the least number of messages. The overhead depends on the network topology, communication model, and noise resilience, as we elaborate upon in the Related Work section below.

Our main result, informally stated as follows, is a deterministic coding scheme that fortifies any asynchronous protocol designed for a noise-free setting over any network topology, such that its resilient simulation withstands the maximal $\Theta(1/n)$-fraction of noise.

▶ **Theorem 1.** *There exists a deterministic coding scheme that takes as an input any asynchronous distributed protocol $\pi$ designed for reliable channels, and outputs an asynchronous distributed protocol $\Pi$ that simulates $\pi$, is resilient to an optimal fraction of $\Theta(1/n)$ of adversarially corrupted messages, and has a multiplicative communication overhead of $O(n \log^2 n)$.*

Our coding scheme introduces a multiplicative overhead of $O(n \log^2 n)$; no other results are known for this model. This overhead should be compared with the state of the art coding scheme by Hoza and Schulman [27]. Their scheme applies to *synchronous* networks where the

topology is *known by all the parties* in advance, and achieves an overhead of $O((|E| \log n)/n)$, where $|E|$ is the number of communication links in the network. Setting the optimal coding overhead in the *asynchronous* model over an arbitrary topology (*unknown to the parties*) remains as an open question.

## 1.1 Techniques

### 1.1.1 A Content-Oblivious BFS Construction

A key ingredient in our coding scheme is a BFS construction which is *content oblivious*. That is, in our BFS construction, the nodes send messages to each other and *ignore their content*, basing their decisions only on the order of received messages. The challenge is to be able to do this despite asynchrony and despite lack of FIFO assumptions. In a sense, our construction can be seen as a variant of the distributed Dijkstra algorithm, with the property that the nodes send "empty messages" that contain no information (alternatively, the nodes ignore the content of received messages).

Recall that the distributed Dijkstra algorithm, see, e.g., [33, Chapter 5], is initiated by some node $r$,[1] which governs the BFS construction layer by layer, where the construction of each layer is called a *phase*. The invariant is that after the $p$-th phase, the algorithm has constructed a BFS tree $T_p$ of depth $p$ rooted at $r$, where all nodes in $T_p$ know their parent and children in $T_p$. The base case is $T_0 = \{r\}$, and the construction of the first layer is as follows. The node $r$ sends an Explore message to all its neighbors, who in turn set $r$ as their parent. Each Explore message is replied to with an Ack message. Once $r$ receives Ack messages from all of its neighbors, the first phase ends and the construction of the second layer begins. Note that $T_1$ indeed holds $r$ and all of its neighbors.

For the $p$-th phase, the root floods a message Phase through $T_{p-1}$. Once a leaf in $T_{p-1}$ receives a Phase message, it sends Explore to all of its neighbors, who in turn set their parent unless already in $T_{p-1}$. Each node that receives an Explore replies with an Ack and an indication of its parent node, so that the exploring node learns which of its neighbors is a child and which is a sibling. Upon receiving an Ack from all of its neighbors, the node sends an Ack to its parent, which propagates it all the way to $r$. Once $r$ has received Ack messages from all of its children, the phase is complete.

Our content-oblivious BFS construction imitates the above behavior while using only a *single* type of message, instead of Phase, Explore and Ack messages. Specifically, the construction begins with $r$ sending a message (Explore[2]) to all of its neighbors, who in turn set $r$ as their parent and reply with a message (Ack). When $r$ receives a message from all of its neighbors, the first phase is complete. Then, $r$ begins the second phase by sending another message (Explore/Phase) to all of its neighbors. This message causes a node that has already set its parent to behave like $r$ – it sends a message to all of its neighbors (Explore) *except for its parent*. After receiving a message (Ack) from all of its neighbors, it sends a message (Ack) to its parent.

One can easily verify that this approach behaves similarly to the Dijkstra algorithm described above, in the sense that every node sets its parent correctly. The only difference is when a node $u$ sends an (Explore) message to its sibling $w$. In the Dijkstra algorithm

[1] In all the protocols we discuss, the root node does not need to be identified in advance. Rather, the algorithm initiates by waking up an *arbitrary* node who will act as the root. From this point an on, nodes wake up when receiving a message from a neighbor.

[2] To ease the readability, we write in parenthesis the functionality of each sent message, but we emphasize that messages in our construction contain no content at all, and the labels of Explore and Ack are given only for the analysis.

the sibling $w$ replies by telling the exploring node $u$ that they are siblings (by indicating the parent of $w$, which is not $u$). However, in our case messages contain no content and $u$ is unable to distinguish whether $w$ is a child or a sibling, since in both cases $w$ should reply to the EXPLORE message in the same way.

Our insight is that serializing each phase provides a solution to the above ambiguity. That is, we let $r$ send a message (EXPLORE/PHASE) to one child at a time, waiting to receive a message (ACK) from that child before sending a message (EXPLORE/PHASE) to the next child. This gives that if a node is expecting a message (EXPLORE) from its parent but instead it receives a message (EXPLORE) from a non-parent neighbor, then it knows that this neighbor must be a sibling. Hence, the node can mark all siblings and distinguish them from its children.

The main advantage of not basing our construction on the content of received messages is that the obtained BFS construction is *inherently tolerant against message corruptions*: the noise has no effect on the construction since the content of the communicated messages is already being ignored. Notice that in our construction, the nodes do not learn their distance from $r$, in contrast to what can easily be obtained in the noise-free case. However, this will suffice for our usage of the BFS tree in our coding scheme.

### 1.1.2 Interactive Coding over Sparse Subgraphs

A crucial framework we rely on in our simulation is a multiparty coding scheme for interactive communication by Hoza and Schulman [27], which is in turn based on ideas from [35]. This coding scheme allows simulating protocols over any graph $G = (V, E)$ and withstands an $O(1/|E|)$-fraction of adversarial message corruption, while incurring a *constant* communication overhead. The caveat of using this scheme for our simulation is that it applies only for *synchronous* protocols that communicate over $G$ in a manner which we call *fully-utilized synchronous*: in each round, every node communicates one symbol over to *each* of its neighbors.

In order to obtain our coding scheme for asynchronous protocol with resilience $\Theta(1/n)$, we first convert the asynchronous input protocol $\pi$ into a fully-utilized synchronous protocol defined over some subgraph $G' = (V, E')$ of $G$ with $|E'| = \Theta(n)$. To this end, we use the BFS tree constructed by our content-oblivious method described above. Once we obtain a BFS tree $\mathcal{T}$, we simulate each message communicated by $\pi$ via $n$ fully-utilized synchronous rounds over the tree $\mathcal{T}$. During each of such $n$ rounds, a message of $\pi$ is flooded throughout $\mathcal{T}$ until it reaches all the nodes and, in particular, its destination node. Note that in every round, all nodes send messages over all the edges of $\mathcal{T}$. This implies a communication overhead of $O(n^2 \log n)$: we have $n$ rounds with $O(n)$ messages per round. The $\log n$ term stems from adding the identity of the source node and the destination node to each flooded message.[3]

Using the Hoza and Schulman [27] coding scheme taking as an input the fully-utilized synchronous protocol defined over the topology $\mathcal{T}$ gives a resilient simulation of $\pi$ which withstands a maximal $\Theta(1/n)$-fraction of corrupted messages. Alas, it is a synchronous simulation, while our environment is asynchronous. Hence, to complete our simulation, we need to use a *synchronizer* [3].

### 1.1.3 A Root-Triggered Synchronizer

In the original error-free setting, if the input protocol to a synchronizer is guaranteed to be fully-utilized then synchronization is trivial. Each node simply attaches a round number to each of its outgoing messages and produces the outgoing messages for round $i + 1$ only after

[3] Throughout this work, all logarithms are taken to base 2.

receiving messages for round $i$ from all of its neighbors. The key difficulty is then for non-fully utilized synchronous input algorithms, in which a node cannot simply wait to receive a message for round $i$ from all of its neighbors, as it may be the case that some of these do not exist.

In our setting, we guarantee that we produce a fully-utilized synchronous algorithm as an input to our synchronizer. However, we do not assume FIFO channels, which means that we cannot rely on the naive synchronizer, despite the promise of a fully-utilized synchronous protocol for an input. Thus, we need a different solution for synchronizing the messages, and our approach is based on having a single node responsible for triggering messages of each round only after the previous round has been simulated by all nodes. To this end, our synchronizer bears similarity to the classic tree-based synchronizer of Awerbuch [3], with the difference that it does not incur any message overhead because it is given a fully-utilized synchronous input.

### 1.1.4 A Spanner-Based Coding Scheme

Finally, we show how to further improve the communication overhead of our coding technique. Routing each message over a tree $\mathcal{T}$ requires $n$ rounds in the worst case for a message to reach its destination. A more efficient solution would be to route each message through a spanning subgraph $S = (V, E_S)$ of $G$ in which the distance over $S$ of every $(u, v) \in E$ is not too large. On the other hand, the Hoza-Schulman coding scheme on $S$ has a noise resilience of $\Theta(1/|E_S|)$, and hence we require $|E_S|$ to be $O(n)$ in order to maintain an optimal resilience level of $\Theta(1/n)$. Luckily, for every $G$ there exist sparse spanning subgraphs in which $|E_S| = O(n)$ while every two neighbors in $G$ are at distance at most $O(\log n)$ in $S$; such subgraphs are known as $O(\log n)$-spanners [33, 34].

Flooding a message of $\pi$ from $u$ to $v$ can be done within $O(\log n)$ rounds, in each of which $O(|E_S|) = O(n)$ messages are sent by a fully-utilized synchronous simulation of $\pi$, leading to our claimed communication overhead of $O(n \log^2 n)$. Here again, the extra $\log n$ term stems from adding identifiers to each flooded message.

However, flooding information over a spanner introduces several other difficulties. For instance, in contrast to the case of a tree, it is not guaranteed anymore that each message arrives only once to its destination – indeed, multiple paths may exist between any two nodes. Furthermore, when multiple nodes send messages, the congestion may cause super-polynomial delays if a simple flooding algorithm is used. Then, due to having multiple paths with arbitrary delays, messages may arrive to their destination out of order. Since the delay is super-polynomial in the worst case, adding a counter to each message increases the overhead by $\omega(\log n)$ and damages the global overhead.

Instead, we provide a contention-resolution flavored technique, which consists of priority-based windows for delivering the messages. In more detail, a message flooding starts only at the beginning of an $O(\log n)$-round window. Multiple messages that are sent during the same window may be dropped during their flooding, yet the source always learns when its message is dropped, so it can retransmit the message in the next window. A similar approach is well-known for constructing a BFS tree when no specific root is given, but our extension of this technique is more involved, since dropped messages *must be resent*.

It remains to explain how to construct the $O(\log n)$-spanner over the noisy network to begin with. For this, we use our previously described tree-based coding scheme to simulate a distributed spanner construction, e.g., the (noiseless) construction of Derbel, Mosbah, and Zemmari [13]. While coding this part incurs a large overhead of $O(n^2 \log n)$, this overhead applies only to the part of constructing the spanner, and the global overhead of our coding scheme is dominated by the overhead of coding the input protocol over the spanner.

## 1.2 Related Work

Performing computations over noisy channels is the heart of *coding for interactive communication*, initiated by Schulman [37, 38]. A long line of work considers the 2-party case in various settings and noise models [10, 6, 23, 17, 14, 29, 25, 20, 7, 9]. See [19] for a survey.

Interactive coding in the multiparty setting was first considered by Rajagopalan and Schulman [35] for the case of stochastic noise. For any topology $G$, they show a coding scheme with an overhead of $O(\log(d+1))$, where $d$ is the maximal degree of $G$. Gelles et al. [22] provide an efficient extension to that scheme. Alon et al. [1] show a coding scheme with an overhead of $O(1)$ for $d$-regular graphs with degree $d = n^{\Omega(1)}$. Braverman et al. [8] demonstrate a lower bound of $\Omega(\log n)$ on the communication over a star graph. All the above works assume fully-utilized synchronous protocols, in which in every round all nodes communicate on all the channels connected to them. Gelles and Kalai [21] show that if nodes are not required to speak at every round, a lower bound of $\Omega(\log n)$ on the overhead can be proved even for graphs with a small degree, e.g., $d = 2$.

Jain et al. [28] show a multiparty coding scheme resilient to an *adversarial* noise fraction of $\Theta(1/n)$ with constant overhead, assuming a topology that contains a star as a subgraph. Lewko and Vitercik [30] improve the communication balance of that scheme. Hoza and Schulman [27] consider fully-utilized synchronous protocols on arbitrary graphs and show a coding with resilience $\Theta(1/|E|)$ and overhead $O(1)$. If the topology of $G$ is known to all nodes, they can route information through a sparser spanning graph with $O(n)$ edges. In this case, they show a coding scheme with an optimal resilience level of $\Theta(1/n)$ and an overhead of $O((|E|\log n)/n)$.

Previous work in distributed settings that allow edge failures are typically different from our setting in various aspects. Most notable are synchrony assumptions, complete communication graphs or addressing specific distributed tasks [36, 39, 24, 40, 12]. Assumptions regarding the noise include random link corruptions [32, 5, 15], or a given bound on the number of links that may exhibit failures [32, 24, 40]. This is in contrast to our work, which addresses an asynchronous setting with an arbitrary topology, and considers the simulation of any distributed task where there is no bound on the number of faulty links. In particular, *all* links may send corrupted messages, with the bound being the number of corruptions rather than the number of faulty links.

Synchronizers for unreliable settings have been studied in [4], which addresses a dynamic setting, and in [26], which assumes faulty nodes.

## 2 Preliminaries

Throughout this work we assume a network described by a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Each node $u \in V$ is a party that participates in the computation and each edge $(u, v) \in E$ is a bi-directional communication channel between nodes $u$ and $v$. The task of the nodes is to conduct some distributed computation given by a deterministic[4] protocol $\pi$, which consists of the algorithm each node (locally) runs. In particular, the protocol dictates to each node which messages to send to which neighbor as a function of all previous communication (and possibly the node's identity, private randomness and private input, if exists). The *communication complexity* of the protocol, $\mathsf{CC}(\pi)$, is the maximal

[4] While we focus here on deterministic protocols, ours result also apply to randomized Monte-Carlo protocols.

number of bits communicated by all nodes in any instance of $\pi$. The *message complexity* of $\pi$ is the maximal number of message sent by all nodes in any instance of $\pi$.

We assume that the topology of $G$ is known only locally, namely, each node $v$ knows only the set $\mathcal{N}_u$ of identities of its own neighbors. However, the size of the network $n$ is known to all nodes.

**Communication Models.** Our protocols are for the *Asynchronous* communication model defined below. In addition, we describe a different communication model named the *Fully-Utilized Synchronous Model*, which is common in previous interactive coding work [35, 27, 1, 8]. In particular, we use coding schemes defined in the fully-utilized synchronous model (specifically, [27]) as primitives for encoding our asynchronous protocols (see Lemma 2 below).

- *Asynchronous Model.* In this setting, there are no timing assumptions. We assume each node is asleep until receiving a message. Once a message is received, the receiver wakes up, performs some local computation, transmits one or more messages to one or more adjacent nodes and goes back to sleep. Messages can be of any length. A protocol starts by waking up a single node $r$ of its choice.
- *The Fully-Utilized Synchronous Model.* Communication in this model works in synchronous *rounds*, determined by a global clock. At every clock tick, every node sends one symbol (from some fixed alphabet $\Sigma$) on each and every one of the communication links connected to it. That is, at every round exactly $2m$ symbols are being communicated.

**Adversarial Channel Noise.** We assume an all-powerful adversary that knows the network $G$, the protocol $\pi$ and the private inputs of the nodes (if there are any). The adversary is able to (a) corrupt messages by changing the content of a transmitted message and (b) rush or delay the delivery of messages by an unbounded but finite amount of time. We restrict the number of messages that the adversary can corrupt, namely, we assume that the adversary can corrupt at most some fixed fraction $\mu$ of the communicated messages. We do not restrict how a message can be corrupted and, in particular, the adversary may replace a sent message $M$ with any other message $M'$ of any length and content. However, our coding scheme will have the invariant that each message contains a single symbol (from a given alphabet $\Sigma$), thus a message corruption will be equivalent to corrupting a single symbol. Note that the adversary is *not* allowed to inject new messages or completely delete existing messages.[5]

**Protocol Simulation, Resilience, and Overhead.** A protocol $\Pi$ is said to *simulate* $\pi$, if after the completion of $\Pi$, each node outputs the transcript it would have seen when running $\pi$ assuming noiseless channels. The protocol $\Pi$ is *resilient* to a $\mu$ fraction of noise, if $\Pi$ succeeds in simulating $\pi$ even if an all powerful adversary completely corrupts up to a fraction $\mu$ of the messages communicated by $\Pi$. The *overhead* of $\Pi$ with respect to $\pi$ is defined by $\mathit{overhead}(\Pi \mid \pi) = \mathsf{CC}(\Pi)/\mathsf{CC}(\pi)$.

A coding scheme $\mathcal{C} : \pi \to \Pi$ converts any input protocol $\pi$ into a resilient version $\Pi = \mathcal{C}(\pi)$. The resilience of a coding scheme is the minimal resilience of any simulation generated by the coding scheme. The (asymptotic) overhead of a coding scheme considers the maximal overhead for the worst input protocol $\pi$ when $\mathsf{CC}(\pi)$ tends to infinity. Namely,

$$\mathit{overhead}(\mathcal{C}) = \limsup_{c\to\infty} \max_{\substack{\pi \text{ s.t.} \\ \mathsf{CC}(\pi)=c}} \mathit{overhead}(\mathcal{C}(\pi) \mid \pi).$$

[5] This type of noise, commonly called, *insertion and deletion* noise is known cause issues of synchronization in the interactive setting [9] and may be destructive for asynchronous protocols [16].

We are mainly interested in how the overhead scales with $n$ and $m$.

A famous multiparty coding scheme in the fully-utilized synchronous model, shown by Hoza and Schulman [27] (based on a previous scheme [35]), provides a coding scheme that simulates any noiseless fully-utilized synchronous protocol $\pi$ defined over some topology $G$ with resilience $\Theta(1/m)$ and a constant overhead $O(1)$.

▶ **Lemma 2** ([27]). *In the fully-utilized synchronous model, any $T$-round protocol $\pi$ can be simulated by a protocol $\Pi = \mathrm{HS}(\pi)$ with round complexity $O(T)$ and communication complexity $O(\mathsf{CC}(\pi))$ that is resilient to adversarial corruption of up to an $\Theta(1/m)$ fraction of the messages.*

## 3 A Distributed Content-Oblivious BFS Algorithm

In this section we show a distributed construction of a BFS tree using messages whose content can be arbitrary. We call this a *content-oblivious* construction. Our algorithm can be seen as a variant of a simple distributed layered-BFS algorithm, see, e.g., [18, 33, 41].

### 3.1 The BFS Algorithm: Description

The BFS construction is initiated by one designated node $r$ we call here the *root*. The construction builds the tree layer by layer. First, the root sends a message to all of its neighbors. This triggers its neighbors to set $r$ as their parent. Each such a neighbor replies a message to $r$ to acknowledge that it has received $r$'s message. Once $r$ has received a message from all of its neighbors, it knows that the first layer is completed, and all nodes with distance 1 have set $r$ as their parent. We call the above an Explore step.

The root then begins a second Explore which causes all nodes at distance 2 to set their parent and connect to the BFS tree. Specifically, the root sends a message to each of its children and waits until all children reply a message to indicate they are done. However, in contrast to previous distributed BFS algorithms, messages are sent *sequentially* – the root sends a message to its next child only after receiving the acknowledgement message from its previous child.

When a node $v$ that has already set its parent $parent_v$ receives a message *from its parent* $parent_v$, it acts as a root and invokes an Explore: it sends a message to all of its neighbors excluding $parent_v$ and waits until they all send a message back. Only then $v$ sends a message to its parent to indicate its Explore process has completed. It is easy to see that when the root completes its $k$-th Explore, all nodes within distance at most $k$ have set their parent and connected to the BFS tree.

A special treatment is needed when a node $u$ receives a message from a node $v$ who is *not* the parent of $u$ during a time at which $u$ is not in the middle of an Explore step. That is, $u$ is not expecting any messages from its neighbors, except for its parent that may trigger it to initiate another Explore step. Recalling that messages are sent to children in a sequential manner, it is easy to verify that such a message delivery may happen only when $v$ has received a message from its own parent and is now processing its own Explore. That is, such a message indicates that $v$ is *a sibling* of $u$ in the BFS tree (namely, $v$ is not a parent nor a child of $u$ in the BFS tree). Thus, upon receiving such a message, $u$ marks $v$ as a sibling and removes it from its list of children. To simplify the presentation, as we elaborate in Remark 1, in next exploration steps $u$ will keep sending messages to $v$ as if it was one of its children.

One additional property that we require from our BFS construction is that all the nodes complete the algorithm *at the same time*. As explained in the introduction, we use this construction as an initial part for our coding scheme. Furthermore, recall that in order to be

**Algorithm 1(a)** Content-oblivious BFS construction: Main Algorithm

**Initialization:** All nodes begin in the INIT state.

For node $r$ designated as root:
**Begin**
  $parent_r \leftarrow \perp$
  $children_r \leftarrow \mathcal{N}_r$
  $count_r \leftarrow 0$
  $state_r \leftarrow$ IDLE

  **while** $state_r \neq$ DONE **do** ▷ Perform $n$ instances of Explore
    $r$ invokes Explore
  **end while**
**End**

noise-resilient, during the BFS construction the nodes ignore the content of the messages and their entire behavior is based on whether or not a message was received. However, once this construction is complete, the nodes send and receive messages according to the coding scheme and it is crucial that a node is able to distinguish messages that belong to the BFS construction from messages of the coding scheme.

We solve this issue by making sure that each node participates in exactly $n$ steps of Explore. Once the node has sent the $n$-th acknowledgement to its parent, the node knows that the next message *from the parent* belongs to the coding scheme rather than to the BFS construction.[6] To make sure that each node participates in exactly $n$ Explore steps, regardless of its distance from $r$, we let every node initiate one additional Explore, which we refer to as a *dummy* Explore. Specifically, when a node completes its $(n-1)$-th Explore, and *before the node sends the acknowledgement back to its parent*, it invokes another Explore step. Now, just by counting the messages received from the parent, every node knows whether the BFS construction has completed or not.

The pseudocode of the BFS construction is given in Algorithm 1(a) and Algorithm 1(b).

## 3.2 The BFS Algorithm: Analysis

In this section we analyze Algorithm 1 and show that it satisfies the following properties.

▶ **Theorem 3.** *For any input $G = (V, E)$ and node $r \in V$, Algorithm 1 finds a BFS tree $\mathcal{T}$ with root $r$. Specifically, each node knows its parent in $T$ and all of its adjacent edges that belong to $\mathcal{T}$. The algorithm communicates $O(nm)$ messages, where no payload is needed in any messages.*

Furthermore, we show that all nodes know that the BFS construction is complete, in the following sense.

▶ **Claim 4.** *At the end of Algorithm 1 all nodes are in state* DONE. *Moreover, if $r$ is in state* DONE *then all other nodes are in state* DONE *as well.*

[6] Note that additional messages may arrive from a sibling node for the BFS construction but still, the next message arriving from the *parent* belongs to the coding scheme rather than the BFS construction.

**Algorithm 1(b)** Content-oblivious BFS construction: Message Handling Procedures

For every node $u$ in state INIT upon receiving a message from node $v$
**procedure** SetParent
  $parent_u \leftarrow v$
  $children_u \leftarrow \mathcal{N}_u \setminus \{v\}$
  $count_u \leftarrow 0$
  $state_u \leftarrow$ IDLE
  send a message to $v$ ▷ an "ACK" message
**end procedure**

For every node $u$ in state IDLE/DONE upon receiving a message from $v \neq parent_u$
**procedure** MarkSibling
  $children_u \leftarrow children_u \setminus \{v\}$
  send a message to $v$ ▷ an "ACK" message
**end procedure**

For every node $u$ in state IDLE upon receiving a message from $parent_u$
**procedure** Explore
  $state_u \leftarrow$ EXPLORE
  $count_u \leftarrow count_u + 1$
  **for all** $v \in \mathcal{N}_u \setminus \{parent_u\}$ **do** ▷ note: **for** is sequential
    send a message to $v$ ▷ an "Explore" message
    wait until a message is received from $v$
  **end for**

  **if** $count_u = n - 1$ **then** ▷ Extra *dummy* Explore
    **for all** $v \in children_u$ **do**
      send a message to $v$
      wait until a message is received from $v$
    **end for**
  **end if**

  send a message to $parent_u$ ▷ an "ACK" message

  **if** $count_u = n - 1$ **then** ▷ Change state to DONE if completed; otherwise, back to IDLE
    $state_u \leftarrow$ DONE
  **else**
    $state_u \leftarrow$ IDLE
  **end if**
**end procedure**

**Proof of Theorem 3.** Let $\mathcal{T}$ be a graph on the nodes $V$ defined at the end of Algorithm 1 in the following manner: If $v = parent_u$, then $(u, v)$ is an edge in $\mathcal{T}$. We begin by proving that $\mathcal{T}$ is a spanning tree. This is implied by the following claim.

▶ **Claim 5.** *At the end of the $k$-th invocation of the root's* Explore *step, all the nodes that are at distance $k$ from $r$ set their parent to a node with distance $k-1$ from $r$ and move to the state* IDLE*, and every node of distance larger than $k$ from $r$ is in state* INIT*.*

**Proof.** We prove the claim by induction on $k$. The base case $k = 1$ follows since in the first Explore invocation all of $r$'s children run SetParent, setting $r$ as their parent, and switch to IDLE. They send message only back to $r$, hence all other nodes remain in INIT.

Assume that the claim holds for the $k$-th invocation and consider the $(k+1)$-th invocation of Explore by $r$. Messages propagating along the BFS tree cause all nodes of distance at most

$k$ to invoke Explore (in some order). This triggers a message to every node of distance $k+1$, which causes it to switch its state to IDLE and set its parent to the first node (of distance $k$) that sent it a message. Note that nodes of distance $k+1$ only communicate back to their parent and do not invoke Explore at this time, so nodes of distance larger than $k+1$ remain in state INIT. At the end of the invocation each Explore, the invoking node switches back to state IDLE. ◀

Next, we prove that each node learns which neighbors are its children and which are not. Assume $(u, v)$ is an edge in $G$ but not in $\mathcal{T}$. We show that at the end of the algorithm $v \notin \mathit{children}_u$ and $u \notin \mathit{children}_v$. Let $t$ be the first time after which both $u$ and $v$ have invoked SetParent. We claim that both $u$ and $v$ invoke Explore after time $t$. This is because time $t$ is within the execution of an Explore step invoked by $r$ and before Line 19 of that execution, and hence for every node $w \neq r$ there is a time $t_w > t$ during the execution of the loop in Lines 19–23 for $r$ in which $w$ invokes Explore.

Finally, we note that since $(u, v)$ is an edge in $G$ but not in $\mathcal{T}$, then neither $u$ is an ancestor of $v$ in $\mathcal{T}$ nor $v$ is an ancestor of $u$ in $\mathcal{T}$. This implies that when $v$ invokes Explore then $u$ is in state IDLE, which causes it to invoke MarkSibling and hence $v \notin \mathit{children}_u$. The proof for $u \notin \mathit{children}_v$ is exactly the same.

Finally let us analyze the message complexity. In Algorithm 1 each node invokes Explore for $n$ times (see also the proof of Claim 4 below), where during each Explore it sends a message on each edge. Therefore, there are $O(n)$ messages sent on each one of the $m$ edges, which amounts to a total message complexity of $O(nm) = O(|V| \cdot |E|)$. ◀

▶ Remark 1. It is possible to reduce the message complexity by sending Explore messages only to $\mathit{children}_v$ nodes. However, this must be delayed at least one Explore step, beyond the point in time where all the neighbors have completed their first Explore (in order to be able to identify siblings). The new message complexity will be $O(|V|^2 + |E|)$. For simplicity, we avoid this optimization and assume Explore messages are sent to all non-parent nodes all the time, incurring a message complexity of $O(|V| \cdot |E|)$.

We now prove Claim 4. This property is important in particular for the next section, as it suggests that there is a point in time (known by the root), when all nodes have completed their BFS algorithm. In hindsight, this allows to distinguish messages that are part of the BFS construction, whose content is ignored, from messages of the coding scheme, whose content is meaningful and must not be ignored.

**Proof of Claim 4.** Note that the Explore procedure works in an DFS manner: a node replies an ACK to its parent only after all of its children reply an ACK to it. Similarly, the root completes an Explore step after receiving an ACK from all its children, which means that they have all completed their Explore steps.

Note that each node invokes exactly $n$ Explore steps due to the dummy Explore step initiated in Line 19. To see this, consider the same algorithm without the extra Explore in Lines 19–23 and note that nodes at distance $k$ from the root $r$ invoke exactly $n-k$ Explore steps. Adding this extra Explore step at every node makes all nodes invoke Explore exactly $n$ times. Specifically, during the $n$-th invocation of Explore by $r$, every node with distance 1 from $r$ invokes its $(n-1)$-th Explore step, and then, *before sending an ACK to* $r$ in Line 30, it invoke its $n$-th Explore step. This then continues in an inductive manner all the way to the leaves.

Only once all of its children have sent an ACK and thus terminated the protocol and switched to DONE, a node replies with an ACK to its parent and changes its state to DONE. It follows that when the root receives an ACK for the $n$-th Explore step from all of its children, all the nodes have terminated the protocol and switched state to DONE. ◀

# 4 A Distributed Interactive Coding Scheme

In this section we show how to simulate any asynchronous protocol over a noisy network whose topology is unknown in advance. Our main theorem for this part is the following.

▶ **Theorem 6.** *Any asynchronous protocol $\pi$ over a network $G$ can be simulated by an asynchronous protocol $\Pi$ resilient to an $\Theta(1/n)$-fraction of adversarial message corruption, and it holds that $\mathsf{CC}(\Pi) = O(nm \log n) + \mathsf{CC}(\pi) \cdot O(n^2 \log n)$.*

## 4.1 A fully-utilized synchronous protocol from an asynchronous input protocol $\pi$

The first ingredient we need is a way to transform an asynchronous protocol (defined over $G$) into a fully-utilized synchronous protocol defined over a given spanning tree $\mathcal{T}$ of $G$.[7] This is done in order to be able to use the Hoza-Schulman coding scheme. This transformation does not need to be robust to noise, as it is not going to be executed as is, but we will rather encode the fully-utilized synchronous protocol and execute the robust version. Later, we transform it back into the asynchronous setting using a synchronizer that is robust to noise.

Recall that in a fully-utilized synchronous protocol nodes operate in rounds, where at each round every node communicates one symbol (from some fixed alphabet $\Sigma$) on each communication channel connected to it. We will assume the alphabet is large enough to convey all the information that our coding scheme needs. In particular, we assume each symbol contains $O(\log n)$ bits.

▶ Remark 2. In the following, we assume the network $G$ is composed of channels with a fixed alphabet $\Sigma$ of size $\text{poly}(n)$. That is, each symbol contains $O(\log n)$ bits.

In order to avoid confusion, we will use the term "symbols" for messages sent by the coding scheme, while using "messages" to indicate the information sent by the noiseless protocol $\pi$.

The construction of our transformation into a fully-utilized synchronous protocol is given in Algorithm 2. In this construction, each node $u$ maintains a queue of symbols that it needs to relay throughout a locally known spanning tree $\mathcal{T}$. The queue is initialized with the bits of any message that $u$ needs to send according to the input protocol $\pi$, where each bit is encapsulated in a symbol that contains the bit value, the identity of the source (i.e., of $u$), and the identity of the destination node. Every symbol received by $u$ is pushed into its queue, and relayed to $u$'s neighbors in future rounds. In particular, upon receiving the symbol $(src, dest, val)$ from a node $w$, the node $u$ pushes the vector $(src, dest, val, w)$ to its queue. If $u$ is the destination node, it does not push the symbol into its queue; instead, $u$ collects this bit for decoding the message.

The transformation works by having each node pop a record from its queue in each round and send the obtained triplet to all of its neighbors in $\mathcal{T}$ except for the node $w$ from which the message was received. If the queue is empty then an empty message is sent to all neighbors in $\mathcal{T}$.

Note that all fragments of a message are received in order at the destination, since $\mathcal{T}$ has no cycles. Therefore, we can assume that the protocol sends a predefined symbol that indicates the end of the message, in order to avoid an assumption of knowledge of the message length. This ensures that Line 17 is well-defined. Our transformation guarantees the following.

[7] The spanning tree $\mathcal{T}$ used here will be later constructed using our content-oblivious BFS construction.

**Algorithm 2** Simulating an asynchronous protocol $\pi$ by a fully-utilized synchronous protocol $\pi'$.

**Initialization:** Given is a BFS tree $\mathcal{T}$ rooted at $r$.

In every round, for every node $u$:
**Begin**
  **for** every node $v$ **do**
    Let $M_1 \cdots M_\ell$ be the bit representation of a message $M$ that $u$ has to send to $v$ in $\pi$.
    Push $(u, v, M_1, \perp), \cdots, (u, v, M_\ell, \perp)$ into $queue_u$
  **end for**
  $(src, dest, val, w) \leftarrow$ pop item out of $queue_u$
  **if** $(src, dest, val, w)$ is not empty **then**
    send $(src, dest, val)$ to every $v \in \mathcal{N}_u(\mathcal{T}) \setminus \{w\}$ and send $\perp$ to $w$
  **else**
    send $\perp$ to every $v \in \mathcal{N}_u(\mathcal{T})$
  **end if**
  For every message $(src, dest, val)$ received from $w$:
  **if** $dest \neq u$ **then**
    push $(src, dest, val, w)$ into $queue_u$
  **else**
    collect the bits $val$ for decoding $M$
  **end if**
**End**

▶ **Lemma 7.** *Algorithm 2 creates a fully-utilized synchronous protocol $\pi'$ that simulates $\pi$, in the sense that all messages of $\pi$ are sent and received. The simulation $\pi'$ has a communication overhead of $O(n^2 \log n)$ with respect to $\pi$, and a message complexity of $\mathsf{CC}(\pi) \cdot O(n^2)$.*

**Proof.** By construction, every node sends a symbol to all of its neighbors in each round and hence Algorithm 2 is a fully-utilized synchronous protocol. In addition, eventually every messages of $\pi$ reaches its destination and hence the obtained fully-utilized synchronous protocol simulates $\pi$. For the communication overhead, note that $O(\log n)$ bits of the identities of source and destination are appended to each bit sent by $\pi$; that is, a symbol size of $O(\log n)$ bits suffices. In addition, a delivery of a single message of $\pi$ may require $O(n)$ rounds of relaying symbols sent along the tree $\mathcal{T}$. In each such round there are $O(n)$ symbols that are sent since the obtained protocol is a fully-utilized synchronous protocol. This implies that $O(n^2)$ symbols are communicated per each bit of $\pi$ and gives a total communication overhead of $O(n^2 \log n)$.

Note that this is a worst-case analysis that assumes a single bit travels within the network at each time so that another bit is sent only after a previous bit reached its destination. If several bits are sent consecutively or if several nodes send bits simultaneously, the resulting number of messages can only decrease. ◀

## 4.2 Root-triggered synchronizers

We now describe our root-triggered synchronizer, which we use in order to execute the resilient synchronous protocol (which can be obtained by using the Hoza-Schulman coding scheme) in our asynchronous setting. We constructed a tree-based synchronizer as in Awerbuch [3].

The synchronizer gets as an input a fully-utilized synchronous protocol $\Pi'$ and outputs an equivalent asynchronous protocol $\Pi$ that simulates $\Pi'$ round by round.

We first describe our simulation of a single round of $\Pi'$ over a *tree*. Our synchronizer works as follows. The protocol begins by waking up an arbitrary node; denote this node as the root. The root initiates the process by sending its messages, determined by $\Pi'$, to its children. This triggers its children to send their messages to their children, but not yet to their parent, and so forth, so that messages propagate all the way to the leaves. Once a leaf receives a message, it sends its message to its parent, and similarly, any node which receives a message from all of its children sends its message to its parent. This continues inductively all the way back to the root, which eventually receives messages from all of its children and complete the simulation of this round of $\Pi'$.

We build upon the above idea in order to simulate a fully-utilized synchronous algorithm $\Pi'$ over an arbitrary graph $S$. That is, each node $u$ has a message $m_{uv}$ designated to each one of its neighbors $v \in \mathcal{N}_u(S)$. [8] The pseudocode is given in Algorithm 3. We single out a node $r$, which we refer to as the *initiator*, which starts by sending a message to all of its neighbors in $S$. This triggers each neighboring node to send its messages to its neighbors, but not yet to its parent, which is now simply the neighbor from which it receives the *first* message. This continues inductively, and only when a node receives messages from all of its neighbors it sends its message to its parent. Eventually, the initiator receives messages from all of its neighbors and completes the simulation of the round.

We prove the following properties of Algorithm 3.

▶ **Lemma 8.** *By the end of Algorithm 3 each node $u$ receives the messages $m_{vu}$ from every node $v \in \mathcal{N}_u(S)$, and all nodes are in state* DONE.

**Proof.** Let $T$ denote the tree rooted at $r$ that is induced by the edges of $S$ that connect each node $u$ with $parent_u$. By construction, each node $u \neq r$ sets its parent to be the first node from which it receives a messages and hence $u$ sets exactly one node as its parent in an acyclic manner, inducing the tree $T$.

We prove by induction on the height of the nodes with respect to $T$, that each node $u$ receives the messages $m_{vu}$ from every node $v \in \mathcal{N}_u(S)$ and then switches its state to DONE. Note that every node sends its messages to all of its neighbors so that eventually all such messages arrive, and we only need to verify that the message from $u$ to $parent_u$ is eventually sent.

The base case is for the leaves of $T$, which indeed receive messages from all of their neighbors since the only messages that get delayed are messages from nodes to their parents. Assume this holds for all nodes at height $h$, and consider a node $u$ at height $h+1$. Node $u$ receives messages from all of its siblings in the tree. By the induction hypothesis, every child $v$ of $u$ in $T$ receives all of its messages and switches to state DONE. This implies that in between, node $v$ sends its message $m_{vu}$ to its parent $u$. When this happens for all nodes $v \in children_u$ it is the case that $u$ receives the messages $m_{vu}$ from every node $v \in \mathcal{N}_u(S)$ and then switches its state to DONE. ◀

By having the initiator $r$ control the simulation of each round of a simulated fully-utilized synchronous protocol $\Pi'$, we obtain synchronization for an arbitrary number of rounds.

[8] Later, in Section 5, we apply our root-triggered synchronizer to an input protocol on $G$ which is fully-utilized on a spanning subgraph $S$ of $G$.

**Algorithm 3** A root-triggered synchronizer for a fully-utilized synchronous protocol $\Pi'$ over a graph $S$.

**Initialization:** All nodes begin in the INIT state.

For node $r$ designated as initiator:
**Begin**
  $state_r \leftarrow$ ACTIVE
  $parent_r \leftarrow \perp$
  $children_r \leftarrow \mathcal{N}_r(S)$

  $r$ sends $m_{rv}$ to each node $v \in children_r$
  $r$ waits to receive a message $m_{vr}$ from every node $v \in children_r$
  $state_r \leftarrow$ DONE
**End**

For every node $u$, upon receiving a message from $w$ when in state INIT:
**Begin**
  $state_u \leftarrow$ ACTIVE
  $parent_u \leftarrow w$
  $children_u \leftarrow \mathcal{N}_r(S) \setminus \{w\}$

  $u$ sends $m_{uv}$ to each node $v \in children_u$
  $u$ waits to receive a message $m_{vu}$ from every node $v \in children_u$
  $u$ sends $m_{uw}$ to $w$
  $state_u \leftarrow$ DONE
**End**

▶ **Corollary 9.** *Multiple consecutive invocations of Algorithm 3 simulate any input fully-utilized synchronous protocol $\Pi'$ round by round, resulting in an asynchronous protocol $\Pi$ that uses the same number of messages.*

## 4.3 The Coding Scheme

We can now complete the details of our coding scheme for asynchronous networks with unknown topology. The scheme consists of two parts. In the first part, the scheme uses the BFS construction given in Section 3 in order to obtain a spanning BFS tree $\mathcal{T}$ of $G$. Note that the nodes ignore the content of messages during this part, therefore an adversary that can only modify messages cannot disturb this part.

In the second part, the scheme translates $\pi$ into a fully-utilized synchronous protocol $\pi'$ via $O(n)$ fully-utilized synchronous rounds over $\mathcal{T}$. This is done using Algorithm 2. The protocol $\pi'$ is still non-resilient to noise and hence is not the protocol that is executed. Instead, we add a coding layer for multiparty interactive communication, namely via the Hoza-Schulman coding scheme, whose properties are given in Lemma 2. This results in a fully-utilized synchronous protocol $\Pi'$ that is resilient to noise, which we then execute through our root-triggered synchronizer to obtain the asynchronous resilient protocol $\Pi$.

The complete construction is given in Algorithm 4. We prove its communication overhead in the following lemma, and then we prove its correctness and resilience.

**Algorithm 4** A coding scheme $\Pi$ for any noiseless asynchronous input protocol $\pi$.

**Initialization:** All nodes begin in the INIT state.

For node $r$ designated as initiator:
**Begin**
    Execute Algorithm 1 with $r$ designated as root. Let $\mathcal{T}$ be the obtained BFS tree.
    Let $\pi'$ be a fully-utilized synchronous algorithm induced by $\pi$ using Algorithm 2.
    Let $\Pi' = \mathrm{HS}(\pi')$ be the Hoza-Schulman coding scheme for $\pi'$.
    Simulate $\Pi'$ using the synchronizer of Algorithm 3 over $\mathcal{T}$ with $r$ as the initiator.
**End**

▶ **Lemma 10.** *For any asynchronous protocol $\pi$ the coding $\Pi$ of Algorithm 4 has a communication complexity of $\mathsf{CC}(\Pi) = O(nm \log n) + \mathsf{CC}(\pi) \cdot O(n^2 \log n)$.*

**Proof.** Recall that we assume channels with a fixed alphabet size, so that each symbol contains $O(\log n)$ bits (Remark 2). The $O(nm \log n)$ term follows from Theorem 3. The transformation of Algorithm 2 induces a communication overhead factor of $O(n^2 \log n)$ per bit of $\pi$, as shown in Lemma 7. By Lemma 2 there exists a resilient fully-utilized synchronous protocol $\Pi'$ that simulates $\pi'$ whose message/communication complexity is linear in the message complexity of $\pi'$. Finally, Corollary 9 gives that the asynchronous simulation of $\Pi'$ via Algorithm 2 has the same message and communication complexity as $\Pi'$. It follows that the total overhead in communication of Algorithm 4 is $O(n^2 \log n)$, as claimed. ◀

▶ Remark 3. Note that the BFS construction (Algorithm 1) ignores the contents of messages. Hence, if we relax the assumption of Remark 2, the communication complexity can be reduced by sending empty messages (without any payload) during that step. In this case the message complexity of $\Pi$ remains $O(mn) + \mathsf{CC}(\pi) \cdot O(n^2)$ yet the communication complexity reduces to $\mathsf{CC}(\Pi) = \mathsf{CC}(\pi) \cdot O(n^2 \log n)$.

▶ Remark 4. In the above, each message sent in $\pi$ is split into single bits and a separate symbols is dedicated to each such bit. However, instead of communicating a single bit $M_i$ in each symbol, nodes can aggregate blocks of $O(\log n)$ bits, so that the payload of each symbol is a single *block* (of $\pi$'s communication) while keeping the coding's symbol size of the magnitude $O(\log n)$.

For some protocols, namely those which send large messages, this may result in a slight logarithmic decrease in the message complexity. This optimization, however, will not change the asymptotic overhead in the worst case, when the protocol $\pi$ communicates a single bit at a time.

▶ **Lemma 11.** *For any asynchronous protocol $\pi$ the coding $\Pi$ of Algorithm 4 correctly simulates $\pi$ even if up to $\Theta(1/n)$ of the messages are adversarially corrupted.*

**Proof.** Correctness and resilience to noise are proved as follows. Theorem 3 proves the correctness of our content-oblivious BFS construction despite noise, since the contents of the sent messages are ignored by the nodes. We emphasize that by Corollary 4, all of the nodes know when to stop ignoring the content of messages for the BFS construction and start executing that synchronizer over $\Pi'$.

Lemma 7 proves that indeed $\pi'$ is a fully-utilized synchronous transformation of $\pi$. By Lemma 2, we have that $\Pi'$ is a fully-utilized synchronous protocol that simulates $\pi'$ in a manner that is resilient to corrupting up to $\Theta(1/|\tilde{E}|)$ of the messages, where $\tilde{E}$ is the edges

over which the protocol communicates. In our case these are the edges of the BFS tree $\mathcal{T}$, and hence this step is resilient to an $\Theta(1/n)$-fraction of corruptions.

Finally, Corollary 9 gives that $\Pi'$ is executed correctly in the asynchronous setting despite noise.

We now need to sum up the maximal number of symbols that can be corrupted and the total number of communicated symbols. Recall that the noise resilience is the ratio between these two sums. Since corruption can only take place on symbols of the Hoza-Schulman coding, of which there are $\mathsf{CC}(\pi) \cdot O(n^2)$ many, we get that the scheme is resilient to at most $O(1/n) \cdot CC(\pi) \cdot O(n^2)$ corrupted symbols. The total number of symbols communicated in the scheme includes also the $O(mn)$ symbols required for constructing the BFS tree, implying that our scheme is resilient to a fraction of symbol corruption equal to

$$\frac{O(1/n) \cdot CC(\pi) \cdot O(n^2)}{O(nm) + CC(\pi) \cdot O(n^2)}.$$

This is asymptotically equal to an $O(1/n)$ fraction of noise when $\mathsf{CC}(\pi) > n$, $\mathsf{CC}(\pi) \to \infty$. ◀

Lemmas 10 and 11 directly give our main theorem for this section, Theorem 6.

## 5 A Spanner-Based Distributed Interactive Coding Scheme

In this section we slightly improve the overhead obtained by the coding scheme of Theorem 6. We demonstrate a family of coding schemes with an interesting tradeoff between their overhead and resilience. The key ingredient is replacing the underlying infrastructure of the BFS tree $\mathcal{T}$ with a sparse spanning graph $S$, where we can trade off the sparseness of the graph (i.e., the number of edges it contains, and as a consequence, the resilience of the obtained coding scheme) with its distance distortion (i.e., the maximal distance in $S$ for any neighboring nodes in $G$, and as a consequence, the added overhead for routing messages through $S$ in the coding scheme).

Assume $u$ sends $v$ a message in the input protocol $\pi$. The coding scheme of Algorithm 4 routes every such message via the BFS tree $\mathcal{T}$. This incurs a delay in $\Pi'$, which can be of $O(n)$ rounds: in the worst case, $u$ and $v$ which are neighbors in $G$ may now be two leaves of $\mathcal{T}$ whose distance is $n$. In fact, even if their distance in $\mathcal{T}$ is smaller, the coding scheme is not aware of this fact and must propagate the message to the entire network. The only guarantee we have in this case is that the message reaches its destination after at most $n$ rounds (of the underlying fully-utilized synchronous protocol).

In this section we suggest a way to reduce the delay factor of $n$ by routing messages over *a spanner* rather than over the tree $\mathcal{T}$.

▶ **Definition 12** ($t$-Spanner)**.** A subgraph $S = (V, E_S)$ is a $t$-spanner of $G = (V, E)$ if for every $(u, v) \in E$ it holds that $dist(u, v) \leq t$ in $S$.

Replacing the BFS tree $\mathcal{T}$ with a $t$-spanner that has $s = |E_S|$ edges ensures that a message reaches its destination after at most $t$ steps (instead of $n$). Since the noise resilience is determined by the number of edges used by the underlying fully-utilized synchronous protocol, by Lemma 2, we obtain a resilience of $\Theta(1/s)$. The main result of this section is the following.

▶ **Theorem 13.** *Let $\pi_{spanner}$ be an asynchronous distributed algorithm for constructing a $t$-spanner $S$ with $s$ edges in a noiseless setting. Any asynchronous protocol $\pi$ over a network $G$ with $\mathsf{CC}(\pi) \gg \mathsf{CC}(\pi_{spanner})$ can be simulated by a noise-resilient asynchronous protocol $\Pi$ resilient to an $\Theta(1/s)$-fraction of message corruption and it holds that $\mathsf{CC}(\Pi) = \mathsf{CC}(\pi) \cdot O(st \log n)$.*

Specifically, due to the existence of $O(\log n)$-spanners with $O(n)$ edges [3, 34] (see also [33, Section 16]), we can let $\pi_{spanner}$ be a distributed construction of a spanner with the same parameters [13] and obtain the following corollary.

▶ **Corollary 14.** *Let $\pi_{spanner}$ be an asynchronous distributed algorithm for constructing a $\log n$-spanner with $O(n)$ edges in a noiseless setting. Any asynchronous protocol $\pi$ over a network $G$ with $\mathsf{CC}(\pi) \gg \mathsf{CC}(\pi_{spanner})$ can be simulated by a noise-resilient asynchronous protocol $\Pi$ resilient to an $\Theta(1/n)$-fraction of message corruption and it holds that $\mathsf{CC}(\Pi) = \mathsf{CC}(\pi) \cdot O(n \log^2 n)$.*

We defer the detailed construction and proofs to the full version of this paper (see [11]).

**Acknowledgement.** We are grateful to Merav Parter for bringing [13] to our attention.

## References


1. Noga Alon, Mark Braverman, Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Reliable communication over highly connected noisy networks. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 165–173. ACM, 2016. doi:10.1145/2933057.2933085.
2. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.
3. Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.
4. Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael E. Saks. Adapting to asynchronous dynamic networks (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 557–570. ACM, 1992. doi:10.1145/129712.129767.
5. Piotr Berman, Krzysztof Diks, and Andrzej Pelc. Reliable broadcasting in logarithmic time with byzantine link failures. *J. Algorithms*, 22(2):199–211, 1997. doi:10.1006/jagm.1996.0810.
6. Zvika Brakerski, Yael Tauman Kalai, and Moni Naor. Fast interactive coding against adversarial noise. *J. ACM*, 61(6):35:1–35:30, 2014. doi:10.1145/2661628.
7. Mark Braverman and Klim Efremenko. List and unique coding for interactive communication in the presence of adversarial noise. *SIAM J. Comput.*, 46(1):388–428, 2017. doi:10.1137/141002001.
8. Mark Braverman, Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Constant-rate coding for multiparty interactive communication is impossible. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 999–1010. ACM, 2016. doi:10.1145/2897518.2897563.
9. Mark Braverman, Ran Gelles, Jieming Mao, and Rafail Ostrovsky. Coding for interactive communication correcting insertions and deletions. *IEEE Trans. Information Theory*, 63(10):6256–6270, 2017. doi:10.1109/TIT.2017.2734881.
10. Mark Braverman and Anup Rao. Toward coding for maximum errors in interactive communication. *IEEE Trans. Information Theory*, 60(11):7248–7255, 2014. doi:10.1109/TIT.2014.2353994.
11. Keren Censor-Hillel, Ran Gelles, and Bernhard Haeupler. Making asynchronous distributed computations robust to noise, 2017. arXiv:1702.07403.

12 Pallab Dasgupta. Agreement under faulty interfaces. *Inf. Process. Lett.*, 65(3):125–129, 1998. doi:10.1016/S0020-0190(97)00202-0.

13 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory Comput. Syst.*, 47(2):368–404, 2010. doi:10.1007/s00224-009-9190-x.

14 Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Maximal noise in interactive communication over erasure channels and channels with feedback. *IEEE Trans. Information Theory*, 62(8):4575–4588, 2016. doi:10.1109/TIT.2016.2582176.

15 Ofer Feinerman, Bernhard Haeupler, and Amos Korman. Breathe before speaking: efficient information dissemination despite noisy, limited and anonymous communication. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 114–123. ACM, 2014. doi:10.1145/2611462.2611469.

16 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

17 Matthew K. Franklin, Ran Gelles, Rafail Ostrovsky, and Leonard J. Schulman. Optimal coding for streaming authentication and interactive communication. *IEEE Trans. Information Theory*, 61(1):133–145, 2015. doi:10.1109/TIT.2014.2367094.

18 Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, M.I.T. Laboratory for Information and Decision Systems, 1982.

19 Ran Gelles. Coding for interactive communication: A survey. *Foundations and Trends in Theoretical Computer Science*, 13(1-2):1–157, 2017. doi:10.1561/0400000079.

20 Ran Gelles, Bernhard Haeupler, Gillat Kol, Noga Ron-Zewi, and Avi Wigderson. Towards optimal deterministic coding for interactive communication. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1922–1936. SIAM, 2016. doi:10.1137/1.9781611974331.ch135.

21 Ran Gelles and Yael Tauman Kalai. Constant-rate interactive coding is impossible, even in constant-degree networks. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPIcs*, pages 21:1–21:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ITCS.2017.21.

22 Ran Gelles, Ankur Moitra, and Amit Sahai. Efficient coding for interactive communication. *IEEE Trans. Information Theory*, 60(3):1899–1913, 2014. doi:10.1109/TIT.2013.2294186.

23 Mohsen Ghaffari, Bernhard Haeupler, and Madhu Sudan. Optimal error rates for interactive coding I: adaptivity and other settings. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 794–803. ACM, 2014. doi:10.1145/2591796.2591872.

24 Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing and Fault Tolerant Systems*, volume 10, pages 139–158. IEEE Computer Society, 1995. URL: http://www.csl.sri.com/papers/dcca95/dcca95.pdf.

25 Bernhard Haeupler. Interactive channel capacity revisited. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 226–235. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.32.

26 Michael Harrington and Arun K. Somani. Synchronizing hypercube networks in the presence of faults. *IEEE Trans. Computers*, 43(10):1175–1183, 1994. doi:10.1109/12.324543.

27 William M. Hoza and Leonard J. Schulman. The adversarial noise threshold for distributed protocols. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 240–258. SIAM, 2016. doi:10.1137/1.9781611974331.ch18.

28 Abhishek Jain, Yael Tauman Kalai, and Allison Bishop Lewko. Interactive coding for multiparty protocols. In Tim Roughgarden, editor, *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 1–10. ACM, 2015. doi:10.1145/2688073.2688109.

29 Gillat Kol and Ran Raz. Interactive channel capacity. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 715–724. ACM, 2013. doi:10.1145/2488608.2488699.

30 Allison Lewko and Ellen Vitercik. Balancing communication for multi-party interactive coding, 2015. arXiv:1503.06381.

31 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

32 Andrzej Pelc. Reliable communication in networks with byzantine link failures. *Networks*, 22(5):441–459, 1992. doi:10.1002/net.3230220503.

33 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

34 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. doi:10.1002/jgt.3190130114.

35 Sridhar Rajagopalan and Leonard J. Schulman. A coding theorem for distributed computation. In Frank Thomson Leighton and Michael T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 790–799. ACM, 1994. doi:10.1145/195058.195462.

36 Hasan Md. Sayeed, Marwan Abu-Amara, and Hosame Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9(3):147–156, 1995. doi:10.1007/s004460050016.

37 Leonard J. Schulman. Communication on noisy channels: A coding theorem for computation. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 724–733. IEEE Computer Society, 1992. doi:10.1109/SFCS.1992.267778.

38 Leonard J. Schulman. Coding for interactive communication. *IEEE Transactions on Information Theory*, 42(6):1745–1756, 1996.

39 Gurdip Singh. Leader election in the presence of link failures. *IEEE Trans. Parallel Distrib. Syst.*, 7(3):231–236, 1996. doi:10.1109/71.491576.

40 Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Trans. Parallel Distrib. Syst.*, 9(4):335–345, 1998. doi:10.1109/71.667895.

41 Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000. Chapter 12.4. Asynchronous BFS Algorithms, pages 414–420.