Robust Decentralized Mean Estimation with Limited Communication*

Gábor Danner[1] and Márk Jelasity[2]

[1] University of Szeged, Hungary
[2] MTA-SZTE Research Group on Artificial Intelligence, and University of Szeged, Hungary

Abstract. Mean estimation, also known as average consensus, is an important computational primitive in decentralized systems. When the average of large vectors has to be computed, as in distributed data mining applications, reducing the communication cost becomes a key design goal. One way of reducing communication cost is to add dynamic stateful encoders and decoders to traditional mean estimation protocols. In this approach, each element of a vector message is encoded in a few bits (often only one bit) and decoded by the recipient node. However, due to this encoding and decoding mechanism, these protocols are much more sensitive to benign failure such as message drop and message delay. Properties such as mass conservation are harder to guarantee. Hence, known approaches are formulated under strong assumptions such as reliable communication, atomic non-overlapping transactions or even full synchrony. In this work, we propose a communication efficient algorithm that supports known codecs even if transactions overlap and the nodes are not synchronized. The algorithm is based on push-pull averaging, with novel features to support fault tolerance and compression. As an independent contribution, we also propose a novel codec, called the pivot codec. We demonstrate experimentally that our algorithm improves the performance of existing codecs and the novel pivot codec dominates the competing codecs in the scenarios we studied.

1 Introduction

Mean estimation has been studied in decentralized computing for a long time [1,6,8,19]. The applications of these algorithms include data fusion in sensor networks [20], distributed control [15] and distributed data mining [17]. A very interesting potential new application is federated learning, where a deep neural network (DNN) model is trained on each node and these models are then averaged centrally [11]. This average computation could be decentralized, allowing for a fully decentralized solution. However, since DNNs may contain millions of floating-point parameters all of which have to be averaged simultaneously, optimizing the utilized bandwidth during decentralized averaging becomes the central problem.

Many approaches have been proposed for bandwidth-efficient average calculation. For example, floating point numbers can be compressed to a few bits using different

* In: Euro-Par 2018, LNCS 11014, pp. 447-–461, 2018. DOI: 10.1007/978-3-319-96983-1_32. This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things")

quantization methods and these quantized values can then be averaged by a server [9, 18]. This is a synchronized and centralized solution, and the approach also introduces an estimation error. Quantization has been studied also in decentralized gossip protocols where the communicated values are quantized onto a fixed discrete range (see, for example, [21]). Here, an approximation error is introduced again, even in reliable networks, and message exchanges cannot overlap in time between any pairs of nodes.

In control theory, more sophisticated dynamic quantization approaches have been proposed that can provide exact convergence at least in reliable systems by compensating for the quantization error. An example is the work of Li et al. [10]. Here, full synchronization and reliability are assumed, and the quantization range is scaled by a fixed scaling function. Dynamic quantization has also been proposed in the context of linear control in general, again, in a synchronized model [5]. Carli et al. [4] adopt the compensating idea in [10] and compare it with other static (non-adaptive) quantization techniques. The same authors also study adaptive quantization; that is, dynamically changing the sensitivity of the quantizer [3] (originally proposed in [2]), which is feasible over a fixed communication overlay. The system model in these studies assumes reliability and atomic communication as well.

A rather different kind of method involves compressing a stream of floating point values using prediction and leading zero count compression [16]. Although this method could be adapted to our application scenario with some modifications, in this study we focus only on the quantization-based compression methods.

Our contributions include a modified push-pull averaging algorithm and a novel codec. These two contributions are orthogonal: the codec can be used along with any algorithm and the push-pull algorithm can use any codec. The novel codec, called *pivot codec*, encodes every floating point value onto a single bit and it can adapt dynamically to the range of the encoded values. The novel push-pull protocol is robust against message drop failure, it does not require the synchronization of the clocks of the nodes, and it includes a smoothing feature based on recorded link-flows that improves the performance of our compression codec. Here, we evaluate our contributions in simulation. We compare our solutions with the competing codecs and algorithms from related work and show that we can improve both robustness and the compression rate significantly.

2 System Model

We model our system as a large set of nodes that communicate via message passing. The protocols we discuss here send very large messages, so the delay of successfully delivered messages is determined by the message size and the available network bandwidth (as opposed to network latency). Our protocols assume that the delay of most (but not necessarily all) of the messages that are delivered is less than an upper bound. This upper bound is at least half of the gossip period, or more, depending on the overlay network. The messages can be lost and their order of delivery is not guaranteed. We do not require time to be synchronized over the nodes but we do assume the existence of a local clock. Each node is assumed to have a small set of neighbors, with which the node can exchange messages. This neighbor set is assumed to be stable and in this study we do not consider node failure. The set of neighbors might be a uniform random sample from the network or it might be defined by any fixed overlay network, depending on the application.

3 Proposed Algorithms

We first discuss our novel codec and then present the modified push-pull averaging protocol in several steps, addressing its robustness, compression, and smoothing features.

3.1 Codec Basics

Central to our algorithms is the concept of encoding and decoding messages over a given directed link using a codec. A codec consists of an encoder and a decoder placed at the origin and the target of the link, respectively. We assume that the link is used to send a series of real valued messages during the execution of the protocol. We follow the notations used in [13]. First of all, the compression (or encoding) is based on quantization, that is, mapping real values to a typically small discrete space (an alphabet) denoted by S. The decoding maps an element of alphabet S back to a real value.

Codecs may also have state. This state might contain, for example, information about the current granularity or scale of the encoding, the previous value transmitted and elapsed time. The state space will be denoted by $\varXi$. Every codec implementation defines its own state space $\varXi$ (if the implementation is stateful). Both the encoder and the decoder are assumed to share the same state space.

We now introduce a notation for the mapping functions mentioned above. Let $Q : \varXi \times \mathbb{R} \to S$ denote the encoder (or quantizer) function that maps a given real value to a quantized encoding based on the current local state of the encoder. Let $K : \varXi \times S \to \mathbb{R}$ denote the decoding function that maps the encoded value back to a real value based on the current local state of the decoder. Finally, let $F : \varXi \times S \to \varXi$ define the state transition function that determines the dynamics of the state of the encoder and the decoder. Note that in a given codec both the encoder and the decoder uses the same F. These three mappings are always executed in tandem, that is, an encoded message is decoded and then the state transition is computed.

Although the encoder and the decoder are two remote agents that communicate over a limited link, the algorithms we discuss will ensure that both of them maintain an identical state. In this sense, we can talk about the state of the codec. To achieve this, first we have to initialize the state using the same value ξ_0. Second, if the encoder and the decoder have identical states at some point in time, then an identical state can be maintained also after the next transmission, because the encoder can simulate the decoder locally, thus they can both execute the state transition function with identical inputs. Note that here we assumed that communication is reliable. If this is not the case, the algorithms using the codec must handle unreliability appropriately so as to maintain the identical states.

3.2 Pivot codec

Here we describe our codec implementation that we coined the *pivot codec*, for reasons that will be explained below. The main goal in our implementation was aggressive compression, so we put only a single bit on the wire for each encoded value. This means $S_{pivot} = \{0, 1\}$.

The intuition behind the design is that we treat the encoder and the decoder as two agents, such that the encoder stores a constant value and the decoder has to guess this value based on a series of encoded messages. Obviously, in real applications the encoded value is rarely constant. However, the design is still efficient if the encoded values do not change much between two transmissions. In fact, this assumption holds in many applications, including decentralized mean approximation, which allows for an efficient compression. Many competing codecs, especially simple quantization techniques, do not make any assumptions about the correlation of subsequent encoded values, hence they are unable to take advantage of the strong positive correlation that is present in many applications.

The codec is stateful. The state is defined by a triple $(\widehat{x}, d, s_{last}) \in \Xi_{pivot} = \mathbb{R} \times \mathbb{R} \times S_{pivot}$. Here, $\widehat{x}$ is the approximation of the *pivotal value*, namely the actual (constant or slowly changing) real value stored by the encoder agent. The remaining values are d, the signed step size, and s_{last}, the last encoded value that was transmitted. The encoding function is given by

$$Q_{pivot}((\widehat{x}, d, s_{last}), x) = \begin{cases} 1, & \text{if } |\widehat{x} + d - x| < |\widehat{x} - x| \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

where x is the value to be encoded. In other words, the encoded value is 1 if and only if adding the current step size to the approximation makes the approximation better. Accordingly, the decoding function

$$K_{pivot}((\widehat{x}, d, s_{last}), s) = \begin{cases} \widehat{x} + d, & \text{if } s = 1 \\ \widehat{x}, & \text{otherwise} \end{cases} \tag{2}$$

will add the step size to the current approximation if and only if a 1 is received. Note that this design ensures that the approximation never gets worse. It can only get better or stay unchanged, assuming the encoded value is a constant. Note that both the encoder and the decoder share the same state. This is possible because the encoder can simulate the decoder locally, thus both the encoder and the decoder can compute the same state transition function given by

$$F_{pivot}((\widehat{x}, d, s_{last}), s) = \begin{cases} (\widehat{x} + d, 2d, s), & \text{if } s = 1 \wedge s_{last} = 1 \\ (\widehat{x} + d, d, s), & \text{if } s = 1 \wedge s_{last} = 0 \\ (\widehat{x}, -d/2, s), & \text{otherwise.} \end{cases} \tag{3}$$

Here, if d is added for the second time, we double it (assuming that the direction is good) and if we have $s = 0$ then we halve the step size and reverse its direction, assuming that adding d overshot the target. The step size is left unchanged after its first successful application (middle line).

In order for the encoder and the decoder to share their state, they also have to be initialized identically. The initial state ξ_0 might use prior knowledge, for example, prior information about the expected mean and the variance of the data are good starting points for $\widehat{x}$ and d, respectively, but a generic value like $\xi_0 = (0, 1, 0)$ can also be used.

Algorithm 1 robust push-pull

```
$x$ is the local approximation of the average, initially the local value to be averaged.
$u_{i,in}$ and $u_{i,out}$ record the number of times the local value was updated as a result of an incoming push or pull message from $i$, respectively.
$s_i$ is the value that was sent in the last push message to $i$.
$\delta_{i,out}$, $\delta_{i,in}$ are the last push, or pull transfers to $i$, respectively.
$id_i$ is the current unique ID created when sending the latest push message to $i$, initially 0.
$id_{max,i}$ is the maximal unique ID received in any push message from $i$, initially $-\infty$.

procedure ONNEXTCYCLE                                        ▷ called every $\Delta$ time units
    $i \leftarrow$ randomOutNeighbor()
    $s_i \leftarrow x$
    $id_i \leftarrow id_i + 1$
    send push message $(u_{i,out}, s_i, id_i)$ to node $i$

procedure ONPUSHMESSAGE($u, s, id, i$)                        ▷ received from node $i$
    if $id_{max,i} < id$ then
        $id_{max,i} \leftarrow id$
        if $u < u_{i,in}$ then        ▷ last pull has not arrived, roll back corresponding update
            $x \leftarrow x + \delta_{i,in}$
            $u_{i,in} \leftarrow u_{i,in} - 1$
        send pull message $(x, id)$ to node $i$
        update($i, in, x, s$)

procedure ONPULLMESSAGE($s, id, i$)                           ▷ received from node $i$
    if $id_i = id$ then
        update($i, out, s_i, s$)

procedure UPDATE($i, d, s_{loc}, s_{rem}$)
    $u_{i,d} \leftarrow u_{i,d} + 1$
    $\delta_{i,d} \leftarrow \eta \cdot \frac{1}{2}(s_{loc} - s_{rem})$
    $x \leftarrow x - \delta_{i,d}$
```

3.3 Robust push-pull averaging

As a first step towards the compressed algorithm, here we propose a variant of push-pull averaging (Algorithm 1) that is robust to message loss and delay and that also allows for the application of codecs later on. We assume that the links are directed. This means that if both $A \rightarrow B$ and $B \rightarrow A$ exist, they are independent links. Over a given directed link there is a series of attempted push-pull exchanges with push messages flowing along the link and the answers (pull messages) moving in the opposite direction. The algorithm ensures that both ends of each link will eventually agree on the flow over the link. This will ensure a sum preservation (also called mass conservation) property which we prove below.

The algorithm is similar to traditional push-pull averaging in that the nodes exchange their values first. However, as a generalization the new value will not be the average of the two values, but instead a difference δ is computed at both sides using a "learning rate" parameter $\eta \in (0, 1]$, where δ can be viewed as the amount of material being transferred by the given push-pull exchange. Note that both sides can compute the same δ (with opposite signs) independently as they both know the two raw values and they have the same parameter η. Here, $\eta = 1$ results in the traditional variant, and smaller values allow for stabilizing convergence when the push-pull exchanges are not atomic, in which case—despite sum-preservation—convergence is not guaranteed.

As for ensuring sum preservation, we assign an increasing unique ID to all push-pull exchanges. Using these IDs we simply drop out-of-order push messages. Dropping push messages has no effect on the update counters and the local approximations so no further repair action is needed. When the pull message arrives in time, the update is performed, and since the sender of the pull message (say, node B) has already performed the same identical update (using the same δ), the state of the network is consistent. If, however, the pull message was dropped or delayed then the update performed by node B has to be rolled back. This is done when B receives the next push message and learns (with the help of the update counters) that its previous pull message had not been received in time. The update can be rolled back using δ, which ensures that the sum in the network is preserved.

After this intuitive explanation, let us describe the sum-preservation property in formal terms. For this, let us assume that there exists a time t after which there are no failures (message drop or delay). We will show that after time t the sum of the approximations will eventually be the same as the original sum of local values.

Definition 1. *We say that, over link $A \rightarrow B$, a* successful transaction *with ID j is completed when node A receives a pull message with $id = j$ from node B before sending the next push message with $id = j + 1$ to B.*

Let j_k be the ID of the kth successful transaction over link $A \rightarrow B$, and let $j_0 = 0$. For any variable v of Algorithm 1, let v^X denote the value of variable v at node X.

Theorem 1. *For any index $K \geq 0$, right after processing the pull message from B to A of a successful transaction j_K (or for $K = 0$ right after initialization), A and B agree on the total amount of mass transferred over the link $A \rightarrow B$, furthermore, $u^A_{B,out} = u^B_{A,in} = K$ holds.*

Proof. The theorem trivially holds for $K = 0$. Assume that the theorem holds for $K = k - 1$. We show that it holds for $K = k$ as well. First of all, line 25 is executed if and only if the transaction is successful. Then, $u^A_{B,out}$ is incremented by 1, therefore $u^A_{B,out} = k$ indeed holds right after the kth successful transaction. As for $u^B_{A,in}$, the inductive assumption states that $u^B_{A,in} = k - 1$ right after the $(k - 1)$-th successful transaction. After this point, there will be a series of incoming push messages that are not out of order with IDs $i_1, \ldots, i_n$ such that $j_{k-1} < i_1 < \cdots < i_n = j_k$, where j_k is the ID of the kth successful transaction. These incoming messages are assumed to be processed sequentially. In all of these push messages we will have $u = k - 1$. It follows that after processing i_1 we will have $u^B_{A,in} = k$ and after processing each new message $i_2, \ldots, i_n$ we will still have $u^B_{A,in} = k$. This means we have $u^A_{B,out} = u^B_{A,in} = k$ right after the successful transaction j_k.

Let us turn to the transferred mass, and show that after the kth successful transaction A and B will add or remove, respectively, the same δ mass from their current approximations. This is analogous to our previous reasoning about the counters $u^A_{B,out}$ and $u^B_{A,in}$, exploiting the observation that only at most one update has to be rolled back between consecutive updates (which can be done due to recording $\delta^B_{A,in}$) until the correct update occurs. Also, due to recording s^A_B both A and B can compute the same δ despite the delay at A between sending the push message and updating after receiving the pull message.

Corollary 1. *After time t push-pull exchanges become atomic transactions so after a new push message is sent on each link, each pair of nodes will agree on the transferred amount of mass, resulting in global mass conservation. Also, the algorithm will become equivalent to the atomic push-pull averaging (for $\eta = 1$), for which convergence has also been shown [6].*

Note that if the message delay is much longer than the gossip period Δ then progress becomes almost impossible, because sending a new push message over a link will often happen sooner than the arrival of the pull message (the reply to the previous push message), so the pull message will be dropped. Therefore, the gossip period should be longer than the average delay. In particular, if the gossip period is at least twice as large as the maximal message delay then no pull messages will be dropped due to delay.

Transactions over different links are allowed to overlap in time. When this happens, it is possible that the variance of the values will temporarily increase, although the sum of the values will remain constant. In networks where transactions overlap to a great degree, it is advisable to set the parameter η to a lower value to increase stability.

3.4 Compressed push-pull averaging

Here, we describe the compressed variant of push-pull averaging, as shown in Algorithm 2. Although the algorithm is very similar to Algorithm 1, we still present the full pseudocode for clarity. Let us first ignore all the f variables. The algorithm is still correct without keeping track of the f values, these are needed to achieve a smoothing effect that we explain later on. Without the f values, the algorithm is best understood as a compressed variant of Algorithm 1 where values are encoded before sending and decoded after reception. There are some small but important additional details that we explain shortly.

In the messages, the value of x is compressed, but the u and id values are not. This is not an issue, however, because our main motivation is the application scenario where x is a large vector of real numbers. The amortized cost of transmitting two uncompressed integers can safely be ignored.

The algorithm works with any codec that is given by the definition of the state space Ξ, the alphabet S, and the functions Q, F and K, as described previously. We maintain a codec for every link and for every direction. That is, for every directed link (j, i) there is a codec for the direction $j \rightarrow i$ as well as $j \leftarrow i$. For the $j \rightarrow i$ direction, node j stores the codec state (used for encoding push messages) in $\xi_{i,out,loc}$ and for the $j \leftarrow i$ direction the codec (used for decoding pull messages) is stored in $\xi_{i,out,rem}$ at node j. In this notation, "out" means that the given codecs are associated with the outgoing link. The states for the incoming links are stored in a similar fashion.

Algorithm 2 compressed smooth push-pull

```
 1: $\xi_{i,in,loc}, \xi_{i,in,rem}, \xi_{i,out,loc}, \xi_{i,out,rem} \in \Xi$ are the states of the codecs for the local node
    and remote node $i$, initially $\xi_0$.
 2: $f_{i,in}, f_{i,out}$ are the amounts of mass transferred so far to $i$, initially 0.
 3: $\xi_{i,in',loc}, \xi_{i,in',rem}$, and $f_{i,in'}$ are the previous values of $\xi_{i,in,loc}, \xi_{i,in,rem}$, and $f_{i,in}$, ini-
    tially $\xi_0, \xi_0$, and 0, respectively.
 4:
 5: **procedure** ONNEXTCYCLE                                   ▷ called every $\Delta$ time units
 6:     $i \leftarrow$ randomOutNeighbor()
 7:     $s_i \leftarrow Q(\xi_{i,out,loc}, x + f_{i,out})$
 8:     $id_i \leftarrow id_i + 1$
 9:     send push message $(u_{i,out}, s_i, id_i)$ to node $i$
10:
11: **procedure** ONPUSHMESSAGE($u, s, id, i$)                        ▷ received from node $i$
12:     **if** $id_{max,i} < id$ **then**
13:         $id_{max,i} \leftarrow id$
14:         **if** $u < u_{i,in}$ **then**        ▷ last pull has not arrived, roll back corresponding update
15:             $x \leftarrow x + \delta_{i,in}$
16:             $u_{i,in} \leftarrow u_{i,in} - 1$
17:             $(\xi_{i,in,loc}, \xi_{i,in,rem}, f_{i,in}) \leftarrow (\xi_{i,in',loc}, \xi_{i,in',rem}, f_{i,in'})$
18:         $s_{pull} \leftarrow Q(\xi_{i,in,loc}, x + f_{i,in})$
19:         $(\xi_{i,in',loc}, \xi_{i,in',rem}, f_{i,in'}) \leftarrow (\xi_{i,in,loc}, \xi_{i,in,rem}, f_{i,in})$
20:         send pull message $(s_{pull}, id)$ to node $i$
21:         update($i, in, s_{pull}, s$)
22:
23: **procedure** ONPULLMESSAGE($s, id, i$)                           ▷ received from node $i$
24:     **if** $id_i = id$ **then**
25:         update($i, out, s_i, s$)
26:
27: **procedure** UPDATE($i, d, s_{loc}, s_{rem}$)
28:     $u_{i,d} \leftarrow u_{i,d} + 1$
29:     $\delta_{i,d} \leftarrow \eta \cdot \frac{1}{2}(K(\xi_{i,d,loc}, s_{loc}) - K(\xi_{i,d,rem}, s_{rem}) - 2f_{i,d})$
30:     $(\xi_{i,d,loc}, \xi_{i,d,rem}, f_{i,d}) \leftarrow (F(\xi_{i,d,loc}, s_{loc}), F(\xi_{i,d,rem}, s_{rem}), f_{i,d} + \delta_{i,d})$
31:     $x \leftarrow x - \delta_{i,d}$
```

Recall that codecs must have identical states at both ends of the link and this state is used for encoding and decoding as well. For example, the codec state $\xi_{i,out,loc}$ at node j for the direction $j \rightarrow i$ should be the same as $\xi_{j,in,rem}$ at node i. This requirement is implemented similarly to the calculation of δ in Algorithm 1. The codec state transitions, too, are calculated at both ends of each link independently, but based on shared information, so both nodes can follow the same state transition path, assuming also that the states have the same initial value ξ_0. This state transition is computed right after computing δ, in line 30.

Apart from δ, here we also need the previous codec states for rolling the last update back if a pull message was dropped or delayed. To this end, the codec states are backed up (line 19) and are rolled back when needed (line 17).

When calculating δ, we must take into account the fact that encoding and decoding typically introduces an error. Therefore, in order to make sure that both nodes compute the same δ, both nodes have to simulate the decoder at the other node, and work with the decoded value instead of the exact value that was sent (line 29). Fortunately, this can be done, since the state of the decoder at the other node can be tracked locally, as explained previously. However, since we are no longer working with the exact values, there is no guarantee that every update will actually reduce variance over the network, so it is advisable to set η to a value less than one.

3.5 Flow compensation

So far we have ignored the f variables in Algorithm 2. The purpose of these variables is to make compression more efficient by making the transmitted values over the same link more similar to each other. This way, good stateful adaptive codecs can adjust their parameters to the right range achieving better compression.

The f values capture the flow over the given link. This approach was inspired by flow-based approaches to averaging to achieve robustness to message loss [7, 14]. However, our goal here is not to achieve robustness, but rather to reduce fluctuations in the transmitted values. The algorithm accumulates these flows for each link in both directions. In addition, the flow value is added to the transmitted value. This has a smoothing effect, because if a large δ value was computed over some link (that is, the value of x changed by a large amount), then the sum of x and the flow will still stay very similar the next time the link is used. The beneficial effect of this on compression will be demonstrated in our experimental study.

Clearly, both nodes can still compute the same δ locally, because the flow value is also known at both ends of a link, only the sign will differ. Hence we can apply the formula in line 29.

4 Simulation Results

We evaluate our algorithms in simulation using PeerSim [12]. Apart from the modified push-pull protocol presented here, we experiment with the synchronized version of average consensus, the most well-known algorithm in related work in connection with quantized communication. In addition, we study a set of codecs and combine these with the two algorithms (synchronized iteration and our push-pull gossip). This way, both the codecs and the algorithms can be compared, as well as their different combinations.

Synchronized average consensus is described, for example, in [1]. The idea in a nutshell is that—assuming the values of the nodes are stored in a vector $x(t)$ at time t—if the adjacency matrix A of the nodes is invertible and doubly stochastic then the iteration $x(t+1) = Ax(t)$ will converge to a vector in which all the elements are equal to the average of the original values. The distributed implementation of such an iteration requires strong synchronization. Quantized and compressed solutions in related work focus on such approaches, as well as slightly more relaxed versions where the adjacency matrix can be different in each iteration, but the different iterations can never overlap.

The codecs we test include simple *floating point quantization (F16, F32)* assuming a floating point representation of 16 and 32 bits (half and single precision, respectively).

Here, the codec is stateless, and decoding is the identity mapping. Encoding involves finding the numerically closest floating point value.

We also include the *zoom in - zoom out codec (Zoom)* of Carli et al. [3]. We cannot present this codec in full detail due to lack of space, but the basic idea is that an m-level quantization is applied such that there is a quantizer mapping to $m-2$ equidistant points within the $[-1, 1]$ interval and the values -1 and 1 are also possible levels used for mapping values that are outside the interval. The codec state also includes a dynamically changing scaling factor that scales this interval according to the values being transferred. This codec resembles the pivot codec we proposed, and to the best of our knowledge this is the state of the art dynamic adaptive codec. Note that the minimal number of quantization levels (or alphabet size) is 3, when $m = 3$. The codec has two additional parameters: $z_{in} \in (0, 1)$ and $z_{out} > 1$. The first determines the zoom-in factor and the second is the zoom-out factor. We fix the setting $z_{out} = 2$ based on the recommendation of the authors and our own preliminary results.

4.1 Experimental setup

The network size is $N = 5{,}000$, and the results are the average of 5 runs. We also simulated a select subset of algorithms with $N = 500{,}000$ (single run) in order to demonstrate scalability. The overlay network is defined by a k-out network, where $k = 5$ or $k = 20$. In the case of synchronized average consensus, we transform this network into a doubly stochastic adjacency matrix A by dropping directionality and setting the weights on the links using the well-known Metropolis-Hastings algorithm: $A_{ij} = 1/(1+\max(d_i, d_j))$, where d_i is the degree of node i. Loop edges are also added with weight $A_{ii} = 1 - \sum_{j \neq i} A_{ij}$.

The initial distribution of values is given by the worst case scenario when one node has a value of 1, and all the other nodes have 0. This way, the true average is $1/N$ (where N is the network size). Our performance metric is the mean squared distance from the true average. We study the mean squared error as a function of the number of bits that are transferred by an average node to average a single value. Recall that we assume that many values are averaged simultaneously (we work with a large vector) so network latency can be ignored. This means that the number of transmitted bits can be converted into wall-clock time if one fixes a common bandwidth value for all the nodes.

We examine the value of the parameter η (see Algorithm 1) using a range depending on the actual codec (we determined the optimal value for each scenario and experimented with neighboring values). We also vary the cycle length Δ. We experiment with short and long cycles. When using short cycles, the round-trip time of a message is assumed to be 98% of the cycle length. With long cycles, the round trip time is assumed to be only 2% of the cycle length. The motivation of looking at these two extreme scenarios is that in the latter case messages overlap to a much lesser extent than in the former case. Thus, we wish to demonstrate that our solutions are robust to short cycles. As for failures, we simulate message drop failure, where the message drop rate is either 0% or 5%.

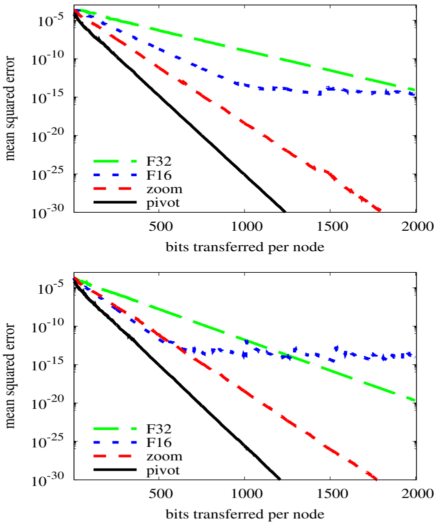

Fig. 1. Comparison of codecs in push-pull with no message drop (left) and a 5% message drop (right) with short cycles (top) and long cycles (bottom). The parameters of all of the codecs have been optimized.

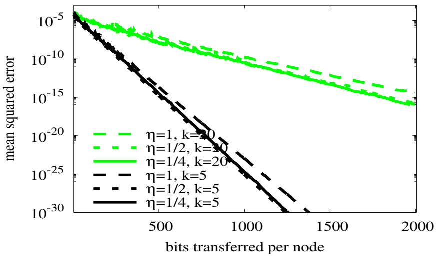

Fig. 2. The effect of parameters η and neighborhood size k on the pivot codec, with no message drop (left) and a 5% message drop (right).

4.2 Results

Figure 1 gives a comparison of the performance of different codecs when using our push-pull algorithm. The parameters were optimized for every codec using a grid search in the space $\eta \in \{2^0, 2^{-1}, \ldots, 2^{-4}\}$, $k \in \{5, 20\}$, $z_{in} \in \{0.35, 0.4, \ldots, 0.85\}$ and $m \in \{4, 8, 16\}$. In all the four scenarios shown on the plots, the best parameter settings were $\eta = 1/2$ and $k = 5$ for the pivot codec and $\eta = 1/4$, $k = 5$, $m = 4$, and $z_{in} = 0.55$ for the zooming codec. For the floating point codecs, $\eta = 1/2$ and $\eta = 1$ were the best for short and long cycles, respectively, and $k = 20$ was the best without message drop. With message drop, the floating point codecs are more stable with $k = 5$ but they converge slightly faster with $k = 20$, especially with short cycles. The pivot codec clearly dominates the other alternatives.

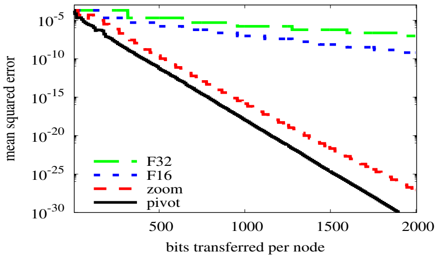

Fig. 3. Comparison of codecs in synchronized average consensus. The parameters of all of the codecs are optimized.

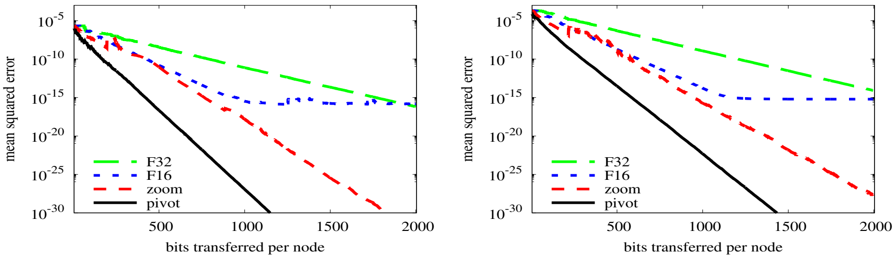

Fig. 4. Comparison of codecs with network size $N = 500{,}000$ (left) and without the flow compensation technique (with $N = 5{,}000$, right).

The difference between $k = 5$ and $k = 20$ is that in the former case more transactions are performed over a given fixed link. In the case of the stateless codecs, this means that $k = 5$ results in a more stable convergence because errors are corrected faster, but with $k = 20$ the correlation between consecutive updates over a fixed link are lower which results in a faster initial convergence. In the case of the pivot codec, Figure 2 illustrates the effect of parameters η and k. It is clear that the algorithm is robust to η, however, parameter k has a significant effect. Unlike the stateless codecs, the pivot codec benefits from a somewhat larger correlation between updates as well as the higher frequency of the updates over a link since these allow for a better prediction of the value at the other end of the link. The zooming codec has a similar behavior (not shown), and we predict that every stateful codec prefers smaller neighborhoods.

Figure 3 presents a similar comparison using the synchronized average consensus algorithm. Note that here, the long and short cycle variants behave identically. Again, the parameters were optimized for every codec and the best parameter settings were $\eta = 1/2$ and $k = 5$ for the pivot codec, $\eta = 1$ and $k = 5$ for the floating point codecs, and $\eta = 1$, $k = 5$, $m = 8$, and $z_{in} = 0.45$ for the zooming codec. Again, the pivot codec dominates the other alternatives. Furthermore, note that, for the pivot codec, the optimal parameters are the same as those in the case of the push-pull algorithm. This suggests that these parameters are robust.

Figures 1 and 3 allow us to compare the push-pull algorithm with the synchronized algorithm. It is clear that all the codecs perform better with push-pull than with the synchronized algorithm. This implies that the push-pull algorithm is a better choice for compression, independently of the selected codec.

Figure 4 contains two remaining observations. First, it demonstrates that the mean squared error of push-pull gossip does not depend on network size as the results with $N = 500{,}000$ (left plot) are very similar to those with $N = 5{,}000$ (Figure 1, top left). This is not surprising as this is predicted by theory when no compression is applied [6].

Second, Figure 4 (right) shows the effect of the flow compensation technique introduced in Algorithm 2, where we used the f variables to smooth the stream of values over each link. As before, we optimized the parameters for all the codecs. The optimal parameter value for the pivot codec turned out to be $\eta = 1/8$ and $k = 5$. This means that if we drastically reduce η, thus smoothing the transactions much more aggressively with this alternative technique, the pivot codec still dominates the other codecs. However, we are not able to get the same compression rate we could achieve with flow compensation (Figure 1) so the flow compensation technique is a valuable addition to the protocol. The other codecs have the same optimal parameters as with flow compensation. Note that the zooming codec also benefits from flow compensation, although to a lesser extent. We also observed that the zooming codec is very sensitive to z_{in} in this case, small deviations from the optimal value result in a dramatic performance loss (not shown).

5 Conclusions

In this paper we presented two contributions, namely a novel push-pull algorithm for computing the average, and a novel codec (called pivot codec) for compressed communication. These two contributions are orthogonal, because the push-pull algorithm can be used with any codec and the pivot codec can be used with any distributed algorithm that supports codecs.

The original features of the push pull algorithm include a mechanism to tolerate message drop failure, and a technique to support overlapping transactions with different neighbors. We also added a mechanism that we called flow compensation, which makes the stream of values over a given link smoother to improve compression. Another smoothing technique is a learning rate parameter η that controls the magnitude of each transaction. The pivot codec that we introduced is based on the intuition that in decentralized aggregation algorithms the values sent over a link are often correlated so compressing the stream is in fact similar to trying to guess a constant value on the other side of an overlay link.

We demonstrated experimentally that the novel codec is superior in the scenarios we studied in terms of the compression rate. We also demonstrated that the flow compensation mechanism indeed improves performance, although the pivot codec dominates the other codecs from related work even without the flow compensation mechanism. We saw that the push-pull protocol is highly robust to overlapping transactions as well, and in general outperforms the synchronized iteration algorithm independently of the codec used.

References

1. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Transactions on Information Theory 52(6), 2508–2530 (2006)
2. Brockett, R.W., Liberzon, D.: Quantized feedback stabilization of linear systems. IEEE Transactions on Automatic Control 45(7), 1279–1289 (2000)
3. Carli, R., Bullo, F., Zampieri, S.: Quantized average consensus via dynamic coding/decoding schemes. Intl. Journal of Robust and Nonlinear Control 20(2), 156–175 (2010)
4. Carli, R., Fagnani, F., Frasca, P., Zampieri, S.: Gossip consensus algorithms via quantized communication. Automatica 46(1), 70–80 (2010)
5. Fu, M., Xie, L.: Finite-level quantized feedback control for linear systems. IEEE Transactions on Automatic Control 54(5), 1165–1170 (2009)
6. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems 23(3), 219–252 (2005)
7. Jesus, P., Baquero, C., Almeida, P.S.: Fault-tolerant aggregation for dynamic networks. In: Proc. 29th IEEE Symposium on Reliable Distributed Systems (SRDS). pp. 37–43 (2010)
8. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. 44th Annual IEEE Symp. on Foundations of Comp. Sci. (FOCS'03) (2003)
9. Konecný, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated learning: Strategies for improving communication efficiency. In: Private Multi-Party Machine Learning (NIPS 2016 Workshop) (2016)
10. Li, T., Fu, M., Xie, L., Zhang, J.F.: Distributed consensus with limited communication data rate. IEEE Transactions on Automatic Control 56(2), 279–292 (2011)
11. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Singh, A., Zhu, J. (eds.) Proc. 20th Intl. Conf. Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 54, pp. 1273–1282 (2017)
12. Montresor, A., Jelasity, M.: Peersim: A scalable P2P simulator. In: Proc. 9th IEEE Intl. Conf. Peer-to-Peer Computing (P2P 2009). pp. 99–100 (2009), extended abstract
13. Nair, G.N., Fagnani, F., Zampieri, S., Evans, R.J.: Feedback control under data rate constraints: An overview. Proc. IEEE 95(1), 108–137 (2007)
14. Niederbrucker, G., Gansterer, W.N.: Robust gossip-based aggregation: A practical point of view. In: Proc. Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 133–147 (2013)
15. Olfati-Saber, R., Fax, J.A., Murray, R.M.: Consensus and cooperation in networked multi-agent systems. Proc. IEEE 95(1), 215–233 (2007)
16. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: Data Compression Conf. (DCC'06). pp. 133–142 (2006)
17. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems 21(2), 164–206 (2003)
18. Suresh, A.T., Yu, F.X., Kumar, S., McMahan, H.B.: Distributed mean estimation with limited communication. In: Proc. 34th Intl. Conf. Machine Learning, (ICML). pp. 3329–3337 (2017)
19. Xiao, L., Boyd, S.: Fast linear iterations for distributed averaging. Systems & Control Letters 53(1), 65–78 (2004)
20. Xiao, L., Boyd, S., Lall, S.: A scheme for robust distributed sensor fusion based on average consensus. In: IPSN'05: Proc. 4th Intl. Symp. on Inf. Proc. in Sensor Networks. p. 9 (2005)
21. Zhu, M., Martinez, S.: On the convergence time of asynchronous distributed quantized averaging algorithms. IEEE Transactions on Automatic Control 56(2), 386–390 (2011)