Tambur: Efficient loss recovery for videoconferencing via streaming codes

Michael Rudow[†], Francis Y. Yan[¶], Abhishek Kumar[†], Ganesh Ananthanarayanan[§], Martin Ellis[§], K.V. Rashmi[†]
[†]Carnegie Mellon University, [¶]Microsoft Research, [§]Microsoft

Abstract

Packet loss degrades the quality of experience (QoE) of videoconferencing. The standard approach to recovering lost packets for long-distance communication where retransmission takes too long is forward error correction (FEC). Conventional approaches for FEC for real-time applications are inefficient at protecting against bursts of losses. Yet such bursts frequently arise in practice and can be better tamed with a new class of theoretical FEC schemes, called "streaming codes," that require significantly less redundancy to recover bursts. However, existing streaming codes do not address the needs of videoconferencing, and their potential to improve the QoE for videoconferencing is largely untested. *Tambur* is a new streaming-codes-based approach to videoconferencing that overcomes the aforementioned limitations. We first evaluate Tambur in simulation over a large corpus of traces from Microsoft Teams. Tambur reduces the frequency of decoding failures for video frames by 26% and the bandwidth used for redundancy by 35% compared to the baseline. We implement Tambur in C++, integrate it with a videoconferencing application, and evaluate end-to-end QoE metrics over an emulated network showcasing substantial benefits for several key metrics. For example, Tambur reduces the frequency and cumulative duration of freezes by 26% and 29%, respectively.

1 Introduction

The quality of videoconferencing calls dictates the effectiveness of remote meetings [17] which are now ubiquitous. Videoconferencing calls can be one-on-one [27] or multi-party [39]. Our work focuses on one-on-one calls. Video quality depends on several key performance indicators, such as freeze, bandwidth, packet loss, and latency [14, 28, 41].

Recovering lost packets is crucial for providing high-quality videoconferencing [33, 48]. Losing even a single packet may prevent rendering a video frame. It may also prohibit rendering multiple future frames (i.e., causing the video to freeze) due to inter-frame dependencies of compressed video. Due to this, it is common for videoconferencing applications to handle packet losses at the application level. The two broad viable solutions are retransmissions and forward error correction (FEC). Both approaches transmit redundant data. Consequently, there is a trade-off between bandwidth allocated for redundancy and transmitting original data. Furthermore, videoconferencing applications must recover lost packets within a strict latency—preferably less than 150 ms [33]—to meet the real-time playback requirement.

Retransmission involves minimal redundant data since it resends only the lost packets. Hence, it is preferred whenever possible [64]. However, retransmission is suitable only for scenarios with short round trip times due to the strict real-time latency requirement of videoconferencing applications. For all other cases, videoconferencing applications rely on FEC to recover lost packets within an acceptable latency.

Block codes are the most common form of FEC employed in production systems today. Under a block code, k "data packets" are used to create r redundant packets—called "parity packets." When some of these $(k+r)$ packets are lost, the k data packets can still be recovered. There are r extra parity packets, so the bandwidth overhead is $(r/k) \times 100\%$. One main objective in designing FEC schemes is to minimize the bandwidth overhead. Common examples of block codes include Reed-Solomon (RS) block codes [55] and fountain (i.e., rateless) codes [40]. Many of the codes, e.g., RS codes, are optimal for *random losses* in which packets are lost independently. For instance, in the above example, if RS codes are used, any k packets suffice for recovery. Hence, block codes are popular for production videoconferencing applications. For example, Microsoft Teams uses RS codes.

Videoconferencing applications send data from compressed video frames over multiple packets. We refer to losing several packets over one or more consecutive frames as a "burst" loss. Burst losses can occur for various reasons, including persistent Wi-Fi interference and network congestion (when applications overflow router buffers and cause correlated losses [29]). Our analysis of packet traces from thousands of video calls from Teams (§3.3) shows that real-world losses faced by videoconferencing applications are indeed bursty.

Block codes are highly inefficient in their bandwidth consumption when recovering from burst losses under real-time latency requirements. In contrast, a relatively new theoretical FEC framework, known as "streaming codes" [5, 42, 43], handle burst losses along with strict latency constraints efficiently. At a high level, streaming codes recover packets lost in a burst sequentially by their respective playback deadlines, whereas block codes recover all the lost packets simultaneously by the earliest playback deadline. Using block codes for loss recovery wastes later parity packets sent before the deadline of the final lost packet. Most prior work on streaming codes is theoretical [5, 20, 22, 26, 35, 36, 42, 43, 59–61], studying bounds and code constructions. A few existing works [6, 25] explore the practical applicability of streaming codes but only for VoIP (i.e., audio, but not video).

Given the dual importance of bandwidth and loss recovery, streaming codes are appealing to videoconferencing applications. However, there are two main challenges. First, there are gaps between existing streaming codes and videoconferencing applications. Most practical variants of streaming codes [6, 25] are limited to settings in which the sizes of the input data are a fixed constant over time. In contrast, in videoconferencing, the sizes of compressed video frames are variable. The only streaming codes that accommodate such variability [59–61] pessimistically assume that a frame is entirely lost or received because the framework involves sending each frame in a single packet. This is seldom true in videoconferencing applications. Existing streaming codes also require that every burst is followed by a *guard space* where all packets are received. But this assumption often does not hold in practice (as we show in §3.2). Also, existing streaming codes set the amount of redundancy with a parameter of a theoretical channel model that is unknown in practice. Second, streaming codes' effectiveness for improving the QoE for real-world videoconferencing applications is untested on real-world data.

This work addresses the aforementioned challenges. We present Tambur, a new communication scheme for bandwidth-efficient loss recovery for videoconferencing.[1] Tambur comprises two components. First, a new streaming code that builds upon a prior theoretical framework [61] while overcoming its limitations with respect to real-world videoconferencing applications. Specifically, Tambur allows for specifying a bandwidth overhead for each frame. Furthermore, for any given bandwidth overhead, Tambur creates data packets and parity packets in a manner that (a) is not overly pessimistic by facilitating recovery from bursts where only some packets are lost per frame and (b) is robust to losses in the guard space. Second, the streaming code is integrated with a machine learning (ML) model to take a predictive decision on the bandwidth allocated to streaming codes. Specifically, a lightweight approach is employed, which uses only a simple model and a single bit of additional feedback.

We analyze packet traces collected from thousands of video calls from Teams and present three key observations in §3: (a) Bursts of packet losses frequently arise. (b) Losses are often, but not always, followed by a guard space of several frames with no losses. (c) Codes employed in production (RS codes) use a significant bandwidth overhead to recover lost packets in real time, depleting the bandwidth for the original data.

We first evaluate Tambur in simulation over a large corpus of traces from Teams (§5.2). We compare Tambur with Teams's FEC ("Block-Within," a block code within a frame; §5.1) and show that Tambur recovers 26.5% more frames with 35.1% less bandwidth overhead.

We also implement and integrate Tambur, several baselines (Block-Within and "Block-Multi," a block code across

[1]Named to convey *Tam*ing *bur*st losses.

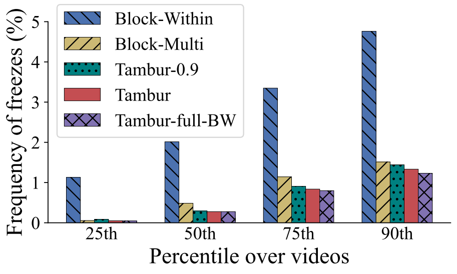
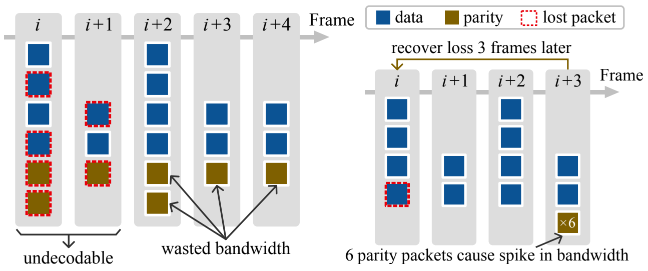


Figure 1: Tambur reduces the ratio of frozen frames to total frames per-video by 78% and 26% compared to Block-Within and Block-Multi, respectively, at a lower bandwidth overhead.

multiple frames) and several variants of Tambur ("Tambur-full-BW," which matches the bandwidth overhead of Block-Within and "Tambur-0.9," which reduces the bandwidth overhead more at the cost of recovering fewer frames) with a videoconferencing benchmark platform that we developed. We then evaluate the schemes over an emulated network to assess the impact on the QoE (§5.4). Tambur, Tambur-full-BW, and Tambur-0.9 reduce the average frequency of video freezes by 26%, 29%, and 17%, respectively, compared with the better of Block-Within and Block-Multi. Fig. 1 shows that these benefits hold across many percentiles. These benefits highlight that Tambur improves the QoE, as it has been shown [44, 53, 66] that video freezes have a detrimental effect on user engagement.

In summary, our main contributions are to:

- Analyze thousands of packet loss logs for video calls taken from a large commercial videoconferencing application, and characterize their suitability for using streaming codes. To the best of our knowledge, this is the first work to evaluate the potential of streaming codes using large-scale, real-world traces.
- Present *Tambur*, which bridges the gap between the theory behind streaming codes and videoconferencing applications by (a) designing a new streaming code that is well-suited to videoconferencing and (b) integrating it with a lightweight ML model to take a predictive decision on the bandwidth allocated to streaming codes.
- Implement a new benchmark platform to enable research on videoconferencing with an easy-to-use interface to integrate and assess new FEC schemes. In addition, implement Tambur, Block-Within, and Block-Multi in C++ and incorporate them into the benchmark platform using the interface.
- Evaluate Tambur over a large corpus of production traces through simulation, and show that it simultaneously reduces the frequency of non-recoverable frames and bandwidth overhead by 26.5% and 35.1%, respectively.
- Evaluate Tambur over emulated networks and show significant improvements over key metrics pertaining to end-to-end QoE (e.g., reducing the frequency of freezes



(a) Within-frame (b) Multi-frame

Figure 2: Two approaches for employing block codes: (a) within each frame and (b) across multiple frames.

by 26% and the cumulative duration of freezes by 29%).

Overall, to the best of our knowledge, this work is the first to establish that streaming codes can improve key metrics relating to the QoE for videoconferencing. This work also showcases the potential of a new form of FEC, streaming codes along with learning-based bandwidth allocation, for bandwidth-efficient loss recovery in videoconferencing. This work poses no ethical issues.

2 Background and motivation

2.1 Conventional FEC and its challenges in videoconferencing

Block codes. One of the most commonly used FECs is the so-called "block codes." The idea of block codes is to encode k data packets, $\langle D[1],\ldots,D[k]\rangle$ to r parity packets into $\langle D[1],\ldots,D[k],P[1],\ldots,P[r]\rangle$, so that the k data packets can be recovered using a subset of the $(k+r)$ packets. When any k of the $(k+r)$ packets suffice for recovery, the block code is termed "maximally distance separable (MDS)." One of the best known examples of MDS codes is the Reed-Solomon (RS) block codes [55]. Other examples of block codes include fountain (i.e., rateless) codes [40], or two-dimensional block codes [67].

Traditionally, FEC applies to packets, but videoconferencing involves transmitting multiple packets for each video frame. One natural solution is to apply a block code to the data packets *within each frame* (Fig. 2a). The parity packets are sent immediately after the final data packet of a frame. A second approach is to apply a block code across the data packets of *multiple frames* (Fig. 2b) by sending all parity packets after the final data packet of the last frame in the block. Our analysis of the production packet loss traces (§3) from Teams shows that the packet losses are bursty. Both approaches have significant limitations for burst losses.

Limitations of block codes for videoconferencing. When packet losses occur as bursts, the within-frame approach wastes the redundancy sent in frames immediately following a burst because it is useless for recovering the lost frames. Although the multi-frame approach overcomes this problem, it has two main drawbacks. First, the latency of recovering losses is high due to waiting for the parity packets, which are sent after the final frame in the block, to recover any packets. The length of the block code must be short lest the latency exceeds the real time deadline to play a frame, leading to an increased bandwidth overhead and reduced robustness to burst losses. Second, packets sent in rapid succession may be lost if a router buffer is full. When a full router buffer coincides with the final frame of a block, no lost packets are recovered.

The bandwidth consumed by parity packets of FEC can be substantially higher than retransmission, even for modest packet loss rates. Unlike retransmission, which only resends lost packets, even an "optimal" FEC scheme does not know which packets will be lost. Hence, it must send far more parity packets than lost packets. For example, to prevent a video freeze, at least one parity packet must be sent every $\approx$ 150ms to cover the scenario of losing a data packet. However, this parity packet is not used if there are no losses.

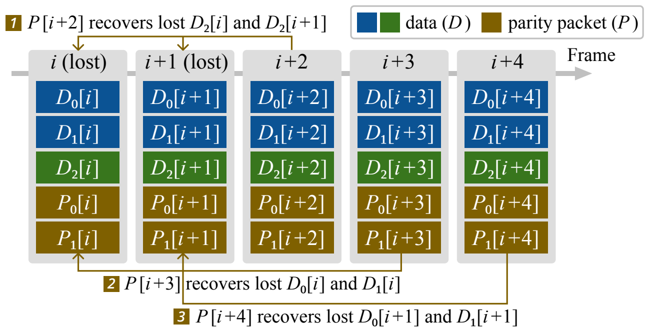
2.2 Streaming codes

A class of codes, known as "streaming codes" [5,42,43,59–61], specifically addresses burst losses and sequential communication between a sender and a receiver. At a high level, streaming codes avoid the limitations of within-frame and multi-frame by (a) sending parity packets with each frame and (b) using all parity packets received by the playback deadline of the *final* frame of a burst for recovering losses. We describe the theoretical framework of streaming codes in detail, provide an illustrative example, and then discuss how it is a promising option for videoconferencing applications impeded by a large gap between the framework and practice.

The streaming codes framework consists of the following. (1) A sender that generates data packets sequentially at regular intervals and transmits packets sequentially to a receiver. (2) An adversarial packet loss channel that introduces burst losses of length b followed by guard spaces of packet receptions. (3) A requirement that the receiver recovers lost data packets within a specified time. The data packet that arrives at the time index i must be recovered by time index $(i+\tau)$. We call the parameter τ the "latency deadline." After a burst, the guard space must be at least τ packets (but longer guard spaces are not needed since bursts are to be recovered within τ packets).

Sequential encoding. The sequential nature of encoding in streaming codes is well suited for videoconferencing, wherein a sequence of compressed frames are to be transmitted periodically (e.g., one every 33.3 ms for a video showing 30 frames per second). We will denote the symbols sent for the ith video frame as $D[i]$, where each symbol can be thought of as a vector of bits.[2] These symbols are distributed over

[2]More formally, a symbol is an element of a mathematical entity called a finite field, and all operations are performed over finite fields using modular arithmetic. For simplicity, readers can just assume the usual arithmetic.



1 $P[i+2]$ recovers lost $D_2[i]$ and $D_2[i+1]$
data (D)
parity packet (P)
i (lost)
$i+1$ (lost)
$i+2$
$i+3$
$i+4$
Frame
$D_0[i]$ $D_0[i+1]$ $D_0[i+2]$ $D_0[i+3]$ $D_0[i+4]$
$D_1[i]$ $D_1[i+1]$ $D_1[i+2]$ $D_1[i+3]$ $D_1[i+4]$
$D_2[i]$ $D_2[i+1]$ $D_2[i+2]$ $D_2[i+3]$ $D_2[i+4]$
$P_0[i]$ $P_0[i+1]$ $P_0[i+2]$ $P_0[i+3]$ $P_0[i+4]$
$P_1[i]$ $P_1[i+1]$ $P_1[i+2]$ $P_1[i+3]$ $P_1[i+4]$
2 $P[i+3]$ recovers lost $D_0[i]$ and $D_1[i]$
3 $P[i+4]$ recovers lost $D_0[i+1]$ and $D_1[i+1]$

Figure 3: Recovering $b = 2$ lost frames starting in frame i within a latency deadline of $\tau = 3$ using a streaming code. For each frame in the burst, all parity symbols sent τ packets later recover its lost symbols.

one or more packets to be sent to the receiver. In addition, some parity symbols, denoted as $P[i]$, are transmitted in one or more packets. These parity symbols are a function (linear combinations) of the data symbols of the past few frames.

Sequential recovery. Under the streaming codes model, the latency deadline parameter τ determines the delay in recovering a lost packet: if $D[j]$ is lost, it must be recovered using symbols from parity packets until $P[j+\tau]$. Each video frame must be recovered within a strict latency to be rendered in real time. The latency deadline parameter τ is set according to the frame rate and one-way delay to induce a suitable maximum latency to recover lost frames. For example, if the maximum tolerable latency is 150 ms (a standard value for real-time video communication [33]), the one-way propagation delay is 50 ms, and a frame is encoded every 33.3 ms (i.e., at 30 fps), τ could be set as 3 $(= (150-50)/33.3)$.

An example of sequential loss recovery of a burst of length 2 starting in frame i within a latency deadline of $\tau = 3$ using existing streaming codes (e.g., [5, 42, 43]) is shown in Fig. 3. Each frame comprises the same amount of data. First, the parity symbols of the packet sent immediately after the burst recovers one-third of the missing data symbols of each lost frame (i.e., $D_2[i]$ and $D_2[i+1]$).Second, the remaining lost data symbols of frames i and $(i+1)$ (i.e., $(D_0[i], D_1[i])$ and $(D_0[i+1], D_1[i+1])$, respectively) are recovered with the parity symbols sent in frames $(i+3)$ and $(i+4)$, respectively.

Streaming codes recover a burst loss by *sequentially recovering* each frame in the burst within its deadline. For a burst loss that encompasses b consecutive frames $\{i, \ldots, i+b-1\}$, a data packet $D[j]$ in the burst is recovered using the parity symbols of $P[i+b], \ldots, P[j+\tau]$. This sequential nature of the recovery of streaming codes allows them to use *all* parity symbols that are received within the deadline. For example, $P[j+\tau]$ is used to recover $D[j]$ *after* the latency deadline of $D[i]$ for $i < j$. In contrast, block codes recover all lost packets together. Hence, the recovery occurs by the first lost frame's deadline (i.e., by the time the symbols of $P[i+\tau]$ are received), wasting the parity symbols sent subsequently. This key difference enables streaming codes to attain significantly lower bandwidth overhead; the longer the burst, the greater the benefits of streaming codes. However, it also requires a guard space of at least τ frames after the burst lest some frame not be recovered with the latency deadline.

2.3 Challenges of using streaming codes for videoconferencing

There are two main challenges in using streaming codes for videoconferencing. First, significant gaps between the theoretical models and practical systems render existing streaming codes incompatible with videoconferencing applications. Second, streaming codes's effectiveness for videoconferencing is untested on large-scale real-world traces. Hence, the potential of streaming codes improving QoE of videoconferencing applications is yet unknown. These challenges are discussed in more detail below.

Gaps between the existing model and videoconferencing applications. The existing practical work on streaming codes [6, 25], like the theoretical work they build upon [5, 42], is limited to settings where the amount of data to be transmitted at each time instant is a fixed constant. However, videoconferencing involves sending compressed video frames whose sizes vary. Only a few streaming code constructions [59–61] can handle this variability. However, as discussed in §2.2, existing streaming codes, including those in [59–61], consider an *adversarial* loss model that imposes bursts of length b. When applied for videoconferencing, the parameter b translates into the number of consecutive frames for which *all* packets are lost. However, videoconferencing applications frequently send multiple packets per frame, and often only some of these packets are lost, as we show in greater detail in §3 for packet loss traces from production. Existing streaming codes are overly pessimistic because they can recover from losing all packets for multiple consecutive frames. This requirement imposes a significant bandwidth penalty, negating the potential bandwidth savings of streaming codes. Streaming codes are also vulnerable to recovery failures if there are *any* losses in the guard space after a burst. But, in practice, many bursts are not followed by such guard spaces (see §3.2).

Applicability of streaming codes in the wild. The benefits of streaming codes for VoIP applications have been studied using simulated losses under theoretical loss models, such as the Gilbert-Elliott channel [23] and over traces [6, 25], wherein each frame is sent in one packet and all frames/packets are of a fixed constant size. However, these results do not apply to videoconferencing applications, which send (a) multiple packets for each frame and (b) varying amounts of data per frame. Streaming codes perform best when each burst occurs across multiple frames and is followed by a guard space of several frames without losses. A natural question is whether such losses arise in videoconferencing and if they can be exploited via streaming codes. To the best of our knowledge, no study of large-scale real-world packet losses establishes the applicability of streaming codes in the wild. Furthermore, establishing that streaming codes are viable to improve the QoE hinges on improving several metrics relating to the QoE. Yet an analysis of streaming codes' impact on such metrics is similarly lacking in the existing literature. Finally, the effect of inter-frame dependencies on the benefits of streaming codes has yet to be assessed, even though inter-frame dependencies are prevalent in videoconferencing.

3 Packet loss in the wild

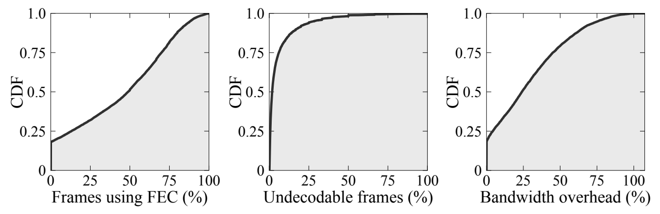
Logs (specifically, packet loss traces) from Microsoft Teams were collected from a random sample of calls over two weeks. One week's traces were held out as a test set for the evaluation.

Teams uses FEC only after a packet loss occurs, which is fairly standard in the industry [64] to avoid wasting bandwidth for the many calls that do not experience any loss. We limit our study to traces with at least two instances of loss since our focus is on improving scenarios *after* FEC is activated (i.e., FEC is turned on after the first loss and then used to recover the second). Our analysis involves approximately 9700 traces, which constitute 16% of all the traces. Studying these traces sheds light on the tail performance, which is crucial for real-world commercial applications. Each trace corresponds to one call and contains the size, sequence number, and send/receive timestamps for each received packet, as well as whether it is a parity packet or data packet; lost packets are identified via missing sequence numbers. Due to the application's data collection method, the traces are limited to the final one minute of the call. Although the logs are for packets, we approximate frame-level information by combining the logs with Teams's packetization logic and have corroborated with the Teams engineers that this approximation is good.

3.1 FEC metrics

Teams employs an RS block code within each frame and varies the bandwidth overhead based on infrequent feedback from the receiver on packet losses. We will denote the FEC scheme used by the application simply as "Block-Within."

We evaluate three metrics over the traces. First, the percent of video frames using FEC for each videoconferencing call (Fig. 4a). The 25th, 50th, and 75th percentile for the percent of video frames over each trace using FEC are 13%, 48.8%, and 70% of calls respectively, indicating that FEC is applied to a significant portion of the frames. Second, the percent of decoding failures for video frames over all frames for each videoconferencing call (Fig. 4b). The 25th, 50th, and 75th percentile for the percent of decoding failures of frames are 0.6%, 1.8%, and 6.1% of calls. Note that the decoding failures should be kept below around 1% to provide high QoE [33].



(a) FEC usage (b) Decoding failures (c) Redundancy

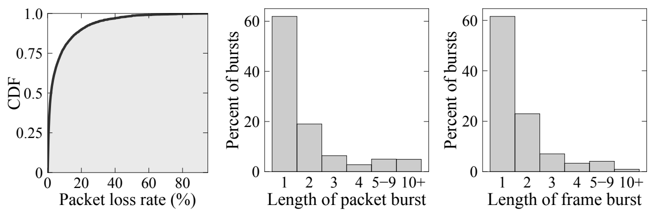
Figure 4: CDFs over the traces from Teams of (a) how often FEC is used to encode frames to protect against packet loss, (b) how often the lost packets are not decoded, and (c) the bandwidth overhead of parity packets.

As such, decoding failures are prevalent enough to tangibly negatively impact the QoE, prompting the need for a more effective FEC mechanism. Third, the bandwidth overhead for each call (Fig. 4c). The 25th, 50th, and 75th percentile for the bandwidth overhead are 4.2%, 24%, and 45% of calls. Thus, reducing the bandwidth overhead will free a significant portion of the bandwidth for these calls.

3.2 Network quality

We analyze the packet losses to assess streaming codes' suitability for real-world videoconferencing applications. To the best of our knowledge, this is the first work to analyze large-scale real-world packet loss traces from this perspective. We analyze three key metrics of losses in Fig. 5. (1) The packet loss rate for each trace (Fig. 5a). (2) The distribution of lengths of bursts of packets measured over all of the calls (Fig. 5b). (3) The distribution of the lengths of bursts of frames (i.e., the number of consecutive frames with at least one packet lost) measured over all of the calls (Fig. 5c). This metric indicates streaming codes' suitability, as they are most effective when bursts of lost packets encompass multiple frames (see § 2.2).

The mean percent of packets lost over the traces is 7%. It is higher than the packet loss in the FCC report [15] since we focus on the traces where FEC is employed. If the other traces from Teams are also considered, the mean packet loss over all traces is 1.7%, which is comparable to the FCC measurement. In earlier studies of end-to-end Internet packet loss, loss rates tended to vary over time and between ISPs and access network technology [8, 18, 24, 54], with ISP queue management policies impacting the loss patterns seen by applications. As discussed in [24], in home broadband networks, loss rates are often less than 1% for long periods, with infrequent periods of very bursty packet loss. Similar patterns are seen in mobile networks, where loss rates tend to increase during handovers [8], and much longer packet loss bursts are seen. Our traces from Teams, described in this section, show similar behavior, with a large number showing very low loss rates, with a long-tail of traces showing extremely bursty packet loss. Specifically, 38.1% of the instances of packet loss in-



(a) Packet loss (b) Packet bursts (c) Frame bursts

Figure 5: Packet loss is prominent (e.g., Fig 5a shows $1-10\%$ packet loss for most traces) and often occurs as bursts across consecutive packets (Fig 5b) or frames (Fig 5c).

volve at least two consecutive packets being lost (Fig. 5b), and 38.4% of instances of packet loss encompass more than one video frame. Such loss patterns can be efficiently recovered by streaming codes (§3.3).

There is also a trade-off between the bandwidth overhead (i.e., the bandwidth used for parity packets) and the probability of decoding failure. The bandwidth overhead cannot be prohibitively high lest there be insufficient bandwidth for the original data. Consequently, the frequency of decoding failures for frames is non-negligible despite using FEC.

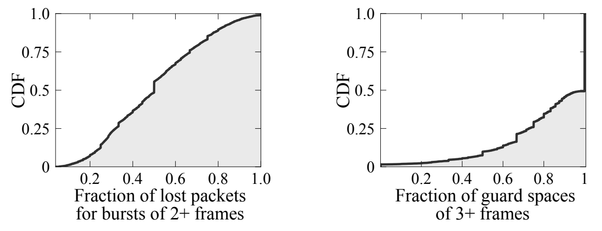
3.3 Potential of streaming codes

Recall from §2.3 that streaming codes are most effective when (a) packet loss occurs as a burst across multiple consecutive frames and (b) the burst loss is followed by a guard space of multiple consecutive frames with no losses. We formalize two metrics to capture these conditions. We then show that the packet losses in the traces exhibit these features.

Measuring bursts. The bandwidth overhead needed to decode a burst depends on the fraction of the packets being lost when losses occur across multiple frames. We introduce a new metric to formalize this notion.

Definition 1 (Multi-frame burstiness) *Suppose a burst occurs across two or more frames, i through j, over which s packets are sent. If l of the s packets are lost, the* ***multi-frame burstiness*** *is defined as l/s.*

For example, suppose Tambur sends packets $(D_0[i], D_1[i], D_2[i])$ and $(D_0[i+1], D_1[i+1])$ over frames i and $(i+1)$, respectively. Suppose $D_1[i], D_2[i]$, and $D_0[i+1]$ are lost. Then the multi-frame burstiness is $3/5$. The multi-frame burstiness is always positive since at least one packet is lost for each frame in the burst. The maximum value of 1 occurs when all packets are lost for all frames in the burst. High values correspond to situations of a high percentage of the packets being lost for multiple consecutive frames. The value of the multi-frame burstiness directly relates to the minimum bandwidth overhead needed for any code to decode lossy frames in real time.



(a) Multi-frame burstiness (b) Guard space sufficiency

Figure 6: The CDFs over the traces of the (a) the multi-frame burstiness (for traces with at least one burst over 2+ frames), and (b) the guard space sufficiency.

Measuring guard spaces. Streaming codes can reduce bandwidth overhead for scenarios where a burst of packet losses is followed by a guard space of at least τ frames that experience reliable transmission, where τ is the latency deadline parameter (§2). We now introduce a new metric to measure the extent to which the guard spaces exhibit this property.

Definition 2 (Guard space sufficiency) *The* ***τ-guard space sufficiency*** *is the fraction of instances in which one or more frames with packet loss are followed by at least τ consecutive frames which experience lossless transmission.*

The value of the guard space sufficiency varies from 0 to 1. It is *negatively* related to the bandwidth overhead needed when using streaming codes. High values for the guard space sufficiency indicate that the bandwidth overhead can be reduced.

Suitability of streaming codes. The multi-frame burstiness and 3-guard space sufficiency is evaluated over the traces in Fig. 6.[3] In Fig. 6a, the value of the multi-frame burstiness is shown to vary over the range 0 to 1, with values at the 25th, 50th, and 75th percentiles of 0.32, 0.5, and 0.67 respectively. This indicates that the bandwidth overhead needed when using streaming codes varies over the traces, as expected. For higher values, more bandwidth must be allocated to redundancy to decode the losses. For lower values, it is possible to make do with less bandwidth used for redundancy. The guard space sufficiency is evaluated over the traces in Fig. 6b, and its values at the 25th, 50th, and 75th percentiles are 0.73, 1.0, and 1.0 respectively. These values imply that *streaming codes are often suitable*. For example, for the traces with a value of 1.0, every single time a burst occurs across multiple frames, streaming codes could have been used to decode the losses with the optimal amount of bandwidth overhead. Yet, the low values indicate insufficient guard spaces for using existing streaming codes to reduce the bandwidth overhead, as they are vulnerable to losses in the guard space.

[3]Recall from § 2.2 that $\tau = 3$ applies for a realistic choice of parameters, in which case a guard space of length 3 is beneficial for loss recovery with streaming codes.

3.4 Key findings

Bursts of packet losses followed by guard spaces arise frequently and are conducive to streaming codes. However, this is not always the case. Bursts are sometimes followed by short guard spaces or involve significant packet loss, in which case the bandwidth overhead cannot be reduced via streaming codes. Hence, integrating streaming codes into real-world applications *requires (a) predicting whether the bandwidth overhead can be reduced without incurring decoding failures, (b) leveraging partial losses in a frame (i.e., losses of only some packets per frame rather than all packets) and (c) adding robustness to losses in the guard space.*

4 Tambur

We present Tambur, which exploits the potential discussed in §3.3 and addresses the challenges discussed in §2.3 by (1) using an ML model to take predictive decisions on the bandwidth overhead, and (2) designing a new streaming code suitable for videoconferencing given any setting for the bandwidth overhead.[4] First, an ML model makes a predictive decision on the number of parity symbols to allocate for each frame. This helps to set the bandwidth overhead to match the network conditions. Second, the parity symbols are defined to provide (a) sequential recovery of bursts over multiple frames while exploiting partial losses, (b) recovery of occasional losses within a single frame immediately, and (c) robustness to a small amount of loss in the guard space after a burst. Third, a new methodology is employed to distribute each frame's data and parity symbols over multiple packets. The design of the parity symbols and their distribution across packets constitute Tambur's streaming code. During loss recovery, Tambur uses the received packets from the frames involved in a burst (i.e., partial losses), which allows for a lower bandwidth overhead than is possible for existing streaming codes that ignore such packets.

Fig. 7 shows how Tambur fits into the stack of a videoconferencing application. The *streaming encoder* encodes data from compressed frames into data packets and parity packets. A *Bandwidth Overhead Predictor* periodically selects bandwidth overhead for each frame using a predictive (ML) model based on the losses observed at the decoder and sends the value to the encoder. The *streaming decoder* uses parity packets to recover lost data packets. We will now describe these components in detail.

4.1 Tambur's streaming code

We present the code in two parts: encoding and decoding.

Encoding. We illustrate how Tambur encodes the ith frame. Fig. 8 shows an example of encoding for $\tau = 3$. The data symbols of this frame, $D[i]$, are sent in data packets, and the parity symbols, $P[i]$, are sent in parity packets. The sizes of the packets are maximized subject to (a) not exceeding an MTU (for example, 1500 bytes in our experiments) to be equal. The previous value of the Bandwidth Overhead Predictor determines how many parity symbols are allocated for frame i. These parity symbols will be sent τ frames later (see "reserved space" in Fig. 8). The number of parity symbols sent for frame i was determined by the size of frame $(i-\tau)$.

[4]The new streaming code builds upon recently developed theoretical streaming codes [60, 61] while overcoming the limitations discussed in §2.3.

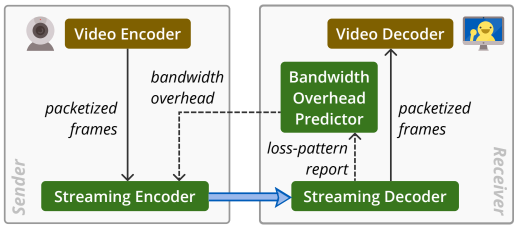


Figure 7: Overview of Tambur. The components in green are specific to Tambur.

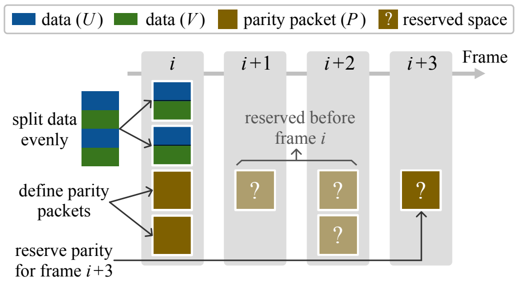


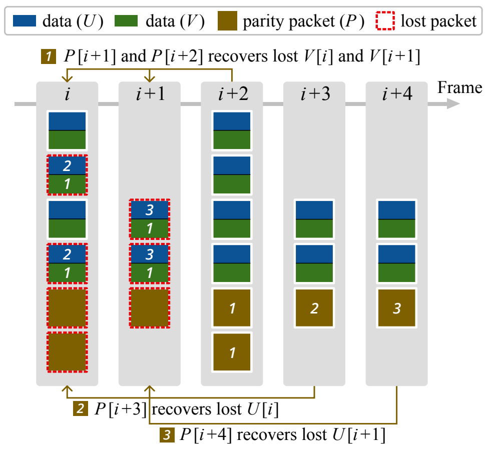
Figure 8: Encoding for $\tau = 3$. Tambur splits frame i evenly into $(V[i], U[i])$ and sends them over data packets. Also, Tambur sends parity packets for recovering $V[i-3], \ldots, V[i], U[i]$ and $U[i-3]$ and reserves space for parity symbols of frame $(i+3)$.

Next, we describe how parity symbols are formed. The symbols of $P[i]$ are linear combinations of the symbols of the $(\tau+1)$ frames, $\{D[i], \ldots, D[i-\tau]\}$. To define the parity symbols, it helps to view the data symbols of the associated $(\tau+1)$ frames as being divided evenly into two parts as $D[j] = (V[j], U[j])$, for $j \in \{i, \ldots, i-\tau\}$. Fig. 8 shows these components in blue and green, respectively.

The symbols of $P[i]$ are carefully designed linear combinations of the symbols of $V[i], \ldots, V[i-\tau], U[i]$, and $U[i-\tau]$. Specifically, $P[i]$ is sum of three quantities: $P[i] := P_1[i] + P_2[i] + P_3[i]$. The symbols of $P_1[i]$ are linear combinations of the symbols of $V[i-\tau], \ldots, V[i-1]$. The symbols of $P_2[i]$ are linear combinations of the symbols of $U[i-\tau]$. The symbols of $P_3[i]$ are linear combinations of the symbols of $U[i]$ and $V[i]$. All linear combinations are carefully chosen to be *linearly independent* linear equations.[5]

Decoding. We describe the decoding process in two parts:

[5]It suffices to take linear equations from three different Cauchy matrices.



data (U)
data (V)
parity packet (P)
lost packet
1 $P[i+1]$ and $P[i+2]$ recovers lost $V[i]$ and $V[i+1]$
i
$i+1$
$i+2$
$i+3$
$i+4$
Frame
2 $P[i+3]$ recovers lost $U[i]$
3 $P[i+4]$ recovers lost $U[i+1]$

Figure 9: Decoding a burst across 2 frames within $\tau = 3$ frames delay using Tambur's streaming code. Data symbols labeled 1, 2, and 3 are decoded using the parity packets with the same label.

(1) occasional packet losses and (2) burst of packet losses. Let all frames before the loss be decoded. First, suppose that packet loss is rare, and the size of $P[i]$ exceeds the number of symbols lost for frame i. Then $P[i]$ suffices to decode the ith frame (specifically, by solving a system of linear equations).

Second, consider a burst of packet losses across two consecutive frames for $\tau = 3$, as is shown in Fig. 9. Packet losses (red-dashed border) span frames i and $(i+1)$. For each frame i, the blue, green, and brown parts represent $U[i]$, $V[i]$, and $P[i]$, respectively. First, $V[i]$ and $V[i+1]$ are both decoded using $P[i+2]$, which consists of independent linear combinations of (a) the symbols of $V[i]$ and $V[i+1]$, and (b) the (received) symbols of $V[i-1]$, $U[i-1]$, $U[i+2]$, and $V[i+2]$. Second, for $j \in \{i, i+1\}$, $U[j]$ is decoded using $P[j+3]$, which consists of independent linear combinations of (a) the symbols of $U[j]$, and (b) the (available) symbols of $V[j-3], \ldots, V[j]$, and $U[j]$. The key to this methodology is that $U[i+1]$ is *not* recovered by the latency deadline of $D[i]$ (i.e., $(i+3)$). This enables using extra parity symbols (i.e., $P[i+4]$) to recover $U[i+1]$ while still decoding each data packet within $\tau = 3$ frames. Appendix A presents the general case. If decoding fails, the receiver queries the sender to generate a new keyframe (i.e., a self-sufficient frame) to handle inter-frame dependencies.

There are three key differences from existing streaming codes for videoconferencing: (1) The data symbols and parity symbols of a frame are sent over multiple packets instead of a single packet. (2) Each frame's parity packets are designed such that they are useful in recovering its lost data packets (in addition to being useful in recovering previously sent frames). (3) The code is flexible enough to allow per-frame bandwidth overhead to be set using the Bandwidth Overhead Predictor.

4.2 Bandwidth overhead predictor

At a high level, Tambur makes use of a predictive model to determine the bandwidth overhead employed by its streaming code (i.e., the amount of "reserved space" in Fig. 8). This predictive model takes as input a feature vector computed by the receiver periodically (e.g., every two seconds), dubbed a *loss-pattern report*. The predictive model's output is then sent to the sender to set the bandwidth overhead for each frame for Tambur's streaming code until the next loss-pattern report is received. For example, a bandwidth overhead of 50% means that if frame i comprises 1000 bytes, 500 bytes of parity symbols are allocated.

Loss-pattern report. Let P be the bitmap of packet losses since the last loss-pattern report, where 1 denotes a loss and 0 is a reception. Let F be a bitmap over all frames since the last loss-pattern report of whether at least one of the frame's packets was lost. The loss-pattern report consists of the following 13 quantities, all of which can be computed in linear time with a single sequential pass over F and P.

- Multi-frame burstiness and guard space sufficiency (§3).
- Fraction of losses for P and F.
- Mean number of consecutive losses for P and F.
- Mean length of guard spaces for P and F.
- Burst density [12] and gap density [12] for P and F[6].
- A score employed by Teams to choose its bandwidth overhead, which is based on the observed fraction of packet losses and lengths of bursts.

Bandwidth overhead prediction via weighted classification. Tambur uses an ML model to determine the bandwidth overhead allocated per frame based on the recent loss conditions. As discussed above in § 4.1, Tambur's streaming code enables such an approach by allowing fine-grained tuning of the bandwidth overhead. To keep the model simple, we select two options for the bandwidth overhead. This approach easily generalizes to more than two values for bandwidth overhead by using a multiclass classifier to enable tuning the bandwidth overhead used by Tambur. In our implementation, we use a small neural network (discussed further in §4.3), although any methodology could be substituted.[7]

The ML model is trained with different *weights* for the two classes based on prioritization of bandwidth savings versus minimizing decoding failures. Essentially, the higher the weight for the class corresponding to the higher bandwidth overhead, the greater the frequency of decoding frames, but the lower the reduction in bandwidth overhead. Videoconferencing service operators can use these weights as a knob to prioritize reducing decoding failures or bandwidth overhead.

Neural network details. Binary classification is conducted using a small fully connected neural network with one hidden layer. The input is the values of the 13 metrics for the

[6]The parameter gMin [12] is set to be 1 and τ for P and F respectively.
[7]We found ML models to outperform heuristics empirically.

previous 3 loss-pattern reports. The cross-entropy loss is applied, and by default the weights for mistakenly reducing the bandwidth overhead (i.e., causing a decoding failure) and not reducing the bandwidth overhead by half (i.e., failing to save bandwidth) are 0.999 and 0.001, respectively. We tested various number of hidden neurons (e.g., 100, 1000, and 10000) and selected 1000 as the smallest option to reach the point of diminishing returns. The model is implemented and trained in PyTorch offline using the traces based on the optimal decision for reducing the bandwidth overhead without causing decoding failures. During inference, it is instantiated in C++.

4.3 Implementation

We implemented Tambur in C++ as part of a new independent library called Tambur that any videoconferencing application can use.[8] At the sender, Tambur takes successive compressed frames as input and outputs data packets and parity packets. At the receiver, Tambur decodes lost packets by solving a system of linear equations using the symbols of the received packets. When packets are lost, we combine properties of streaming codes with an open-source min-cut/max-flow algorithm [11] to determine which data symbols can be decoded using which parity symbols in negligible time (see Appendix B). Data is then decoded by solving the smallest full-rank systems of linear equations.

We use a small header to provide frame-level information needed for decoding. This includes sequence numbers for packets and frames and relative positions of a packet within a frame and amongst parity packets. The streaming decoder also needs the size of the lost frame in order to decode it (even when all packets corresponding to the frame are lost); hence, we encode the sequence of frame sizes using a streaming code and send one parity symbol of this code in each packet.

The library provides an interface for rapidly prototyping new FEC schemes. We used this interface to implement the baselines from §1 (i.e., Block-Within and Block-Multi).

The core arithmetic of linear encoding and solving a system of linear equations for decoding is done using Jerasure 2.0 [52], an open-source library in C/C++ with modules for key operations of erasure coding. Jerasure 2.0 is built on top of the GF-Complete library [51], which uses Intel SIMD instructions to perform Galois Field arithmetic quickly. Tambur involves encoding data into "coding blocks" of 256 bytes, each of which uses the same linear equations. Extending Tambur to use hardware offload to encode and decode frames is a potential avenue of future work.

Integration with videoconferencing. To validate Tambur's effectiveness in the real world, we integrate it with *Ringmaster*[9], a newvideoconferencing platform that emulates one-on-one video calls for benchmarking FEC schemes. Ringmaster is implemented in ∼4000 lines of C++. Its video sender reads raw frames from an input Y4M video file on disk at a precise frame rate (e.g., 30 fps) and compresses them with the VP9 encoder in the `libvpx` [1] library using similar codec configuration as in WebRTC [2]. A user-provided FEC scheme provides parity data for the encoded frames, which is sent over UDP after packetization to the video receiver. Upon receiving the frames, the video receiver applies the FEC decoder and VP9 decoder sequentially to decode and render the original video frames. In addition, Ringmaster allows for requesting new keyframes, e.g., when the receiver fails to recover a video frame due to excessive loss of packets and thus requests the sender to encode a new keyframe so as to resume the video. At the end of the automated call, QoE metrics are computed by aggregating logs from both endpoints, which record the timestamps when each frame is encoded or decoded, along with its frame ID, size, FEC bandwidth overhead, etc.

Ringmaster provides clean and modular interfaces that we use to integrate it into Tambur. Combining Ringmaster with Tambur enables benchmarks of FEC schemes'performance featuring QoE metrics, e.g., video freezes, per-frame delay, rendered frame rate, for FEC schemes implemented via Tambur's interface. Furthermore, Ringmaster also allows researchers to isolate the impact of FEC and disable modules that interfere with FEC, such as bandwidth estimation [13] and packet retransmission.

[8]https://github.com/Thesys-lab/tambur
[9]https://github.com/microsoft/ringmaster

5 Evaluation

To assess whether Tambur can improve the QoE, we ask:

- Can Tambur provide significant benefits for metrics relating to FEC on real-world losses?
- Do the benefits of Tambur lead to a higher QoE?

5.1 Experimental methodology and highlights

Videoconferencing application parameters. In our experiments, we aim for a maximum tolerable latency of 150 ms to meet industry recommendations [33], which is a fairly standard value for interactive video. The frame rate is taken to be 30 fps, which is a typical value in videoconferencing. The inter-frame arrival time for 30 fps is 33.3ms. Allowing for a one-way frame delay of 50 ms leaves room for a decoding delay of around 100ms. Thus, the parameter τ can be at most 3 (frames) for the end-to-end latency (i.e., $33.3\tau+50$) to be at most ≈ 150 ms. The two options for the bandwidth overhead of Tambur are to match or use half of the bandwidth overhead of the baseline coding scheme, Block-Within, which is introduced next.

Coding schemes. We evaluate six coding schemes. (1) **Block-Within** (Fig. 2a), which applies RS codes within a frame. This scheme is employed in production by Teams. (2) **Block-Multi** (Fig. 2b) which applies RS codes across $(\tau+1)=4$

frames. RS codes are *optimal* block codes, and hence the above two baselines outperform other block codes such as fountain or rateless codes in recovering losses and bandwidth overhead. (3) **Tambur-full-BW**, which is a variant of Tambur that matches Block-Within's bandwidth overhead. (4) **Tambur-0.9**, which is Tambur with the neural network trained to prioritize bandwidth savings more by decreasing the weight of misclassification from 0.999 to 0.9 in the loss function. Thus, Tambur-0.9 prioritizes reducing the bandwidth overhead more than Tambur. (5) **Tambur-low-BW**, which is a variant of Tambur that uses 50% of the bandwidth overhead of Block-Within. (6) **Oracle**, which optimally selects between Tambur-full-BW, Tambur-low-BW, or Block-Within. Each time the sender obtains feedback from the receiver, the Oracle selects the scheme with the smallest bandwidth overhead among the scheme(s) that recover the most frames. This choice never causes a non-recoverable loss. Consequently, the Oracle always recovers at least as many frames as Block-Within, Tambur, and Tambur-full-BW. The bandwidth overhead of Block-Within and Block-Multi is never reduced to ensure a fair comparison of Tambur's loss recovery capabilities and because both baselines already perform worse than Tambur despite using the full bandwidth overhead. Like Tambur, Block-Within and Block-Multi send feedback to the sender once FEC decoding has failed to trigger a new keyframe as a fallback mechanism to handle inter-frame dependencies.

Metrics. We evaluate the following metrics: (1) Percent of non-recoverable frames, which is the percentage of compressed frames that are not recovered. (2) Bandwidth overhead for FEC. (3) Percent of non-rendered frames, which is the percent of frames that are not played by the receiver—this includes non-recovered frames and recovered frames that depend on non-recovered frames. (4) Latency, which is the duration between a frame being created and rendered. (5) Frequency of freezes, which is the number of times the receiver's video is frozen. (6) Duration of freezes, which is the cumulative length of time where the receiver's video is frozen.[10] We calculate these metrics only for the frames where FEC is applied (i.e., where FEC affects the quality). We compute one value per call (e.g., median duration of freezes, bandwidth overhead, etc.) and then consider the percentiles over these values. For latency, we consider all frames over all calls.

QoE is difficult to measure precisely with so-called "QoE models" [68] because it depends on video-specific properties (e.g., in sports, video quality during gameplay matters more than during timeouts). But several works [7, 21, 37] have shown that key metrics for QoE (e.g., frequency of freezes, duration of freezes, bandwidth, etc.) impact the mean opinion score—the gold standard measure of QoE. These metrics also affect user interactions (e.g., users watch more video when there are fewer freezes). In fact, [19] showed that cumulative freeze duration is crucial for QoE, as well as the importance of bitrate and frequency of video freezes for live video.

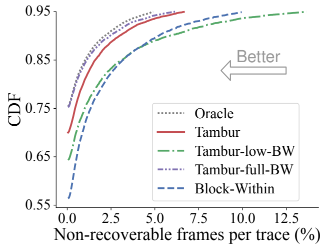
Offline evaluation. We evaluate the performance of Block-Within, Tambur, Tambur-full-BW, Tambur-0.9, Tambur-low-BW, and Oracle over the test set of traces from Teams described in §3, which was held out from the previous analyses. The packet logs provide the performance of Block-Within. We make two safe assumptions to evaluate the remaining schemes over the traces: (a) modifying the payload of a packet, but not its size, would not change whether it is lost or received; (b) reducing the size of a packet's payloads would not incur any new packet losses. Each data packet is sent identically as in the trace, the payloads for the parity packets are changed, the sizes of the parity packets are sometimes reduced, and the bitmap of packet losses from the trace is used. To satisfy the assumptions, we must force Tambur to send the number of parity symbols allocated for each frame within the frame (rather than delayed by τ frames), which we expect to *degrade Tambur's performance*. This enforcement alters the number of parity packets sent under Tambur but not how their symbols are defined. Block-Multi is excluded because it sends all parity packets after the final data packet of the final frame of the block, so its performance cannot be fairly simulated using the production traces.

Online evaluation. We evaluate prototype implementations of Block-Within, Block-Multi, Tambur, Tambur-full-BW, and Tambur-0.9 integrated with Ringmaster (the videoconferencing benchmark platform described in §4.3) via network emulation using Mahimahi [46] while simulating a Gilbert-Elliott (GE) [23] loss model over a dataset of 80 videos. Specifically, we evaluate 20 video calls from [16, 47] at four constant bitrates each (namely, 500, 1000, 1500, and 2000 kbps) to isolate the effect of FEC. The bandwidth overhead is set to 50% for Block-Within (likewise, for Block-Multi and Tambur-full-BW).[11] The GE loss model is a standard loss model which is a Markov model with two states: "good" and "bad," each with an associated probability of packet loss. For a fair and realistic comparison, different coding schemes must experience the same distribution of burst losses at the frame level even though they send differing numbers of packets per frame. Therefore, we consider transitions between the states occurring once at the start of every frame (i.e., once every 33.3 ms) rather than a transition between states every packet, which is commonly used in the literature when only one packet is sent per frame. Packets within each frame are lost independently with the same probability. The modified GE channel can be viewed as a buffer overflowing for a short period, as can arise from on/off characteristics of traffic [49]. Appendix C details how we set the parameters of the GE model based on the losses from the traces.

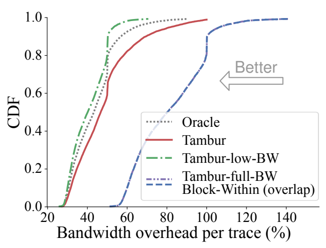
Result highlights.

[10] We use the definition of freezes and duration of freezes from the most recent (unofficial) draft of identifiers for WebRTC's statistics [10].

[11] The bandwidth overhead is sometimes slightly higher for *all* schemes due to rounding and ensuring at least one parity packet is sent per frame.



(a) Non-recoverable frames



(b) Bandwidth overhead

Figure 10: CDFs for the percent of non-recoverable frames for the 55th through 95th percentiles and the bandwidth overhead for the offline evaluation.

- In offline evaluation, Tambur reduces the frequency of non-recoverable frames by 26.5% while using 35.1% less bandwidth overhead.
- In online evaluation, Tambur reduces frequency of non-rendered frames, frequency of freezes, and duration of freezes by 28%, 26%, and 29%, respectively compared to Block-Multi, and by 73%, 78%, and 77% compared to those of Block-Within. Block-Multi has a significantly higher latency than Block-Within (see Fig. 13b).
- Modest memory overhead and median encoding and decoding times of 575 KB, 1.7ms, and 3.4ms, respectively.

5.2 Offline evaluation

We assess only the frequency of non-recoverable frames and the bandwidth overhead for offline traces because the remaining metrics are unavailable. Fig. 10a shows the CDF of the percent of non-recoverable frames from 55th to 95th percentiles over the traces. These percentiles correspond roughly to the 92nd to 99th percentile over all traces. The Oracle reduces the total number of non-recoverable frames by 44.2% compared to Block-Within and reflects an upper bound on performance. Tambur-full-BW reduces the frequency of non-recoverable frames by 33% compared to Block-Within, indicating the potential improvements of using streaming codes. In contrast, Tambur-low-BW *increases* the frequency of non-recoverable frames by 34.7% compared to Block-Within, indicating the need for more sophisticated methods to reduce the bandwidth overhead without incurring a significant penalty in non-recoverable frames. By using a predictive model to determine the bandwidth overhead, Tambur reduces the bandwidth overhead by 35% while simultaneously reducing the number of non-recoverable frames by 26.5% compared to Block-Within (Fig. 10b). §5.3 summarizes the spectrum of bandwidth savings versus recovering frames for Tambur based on tuning the associated weight parameter.

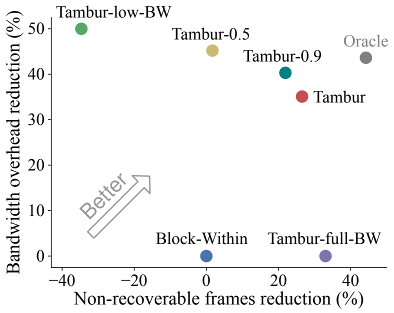


Figure 11: Sensitivity analysis of the weights for the classes used in the predictive model for the frequency of non-recoverable frames and bandwidth overhead over all of the frames where FEC is used in the traces.

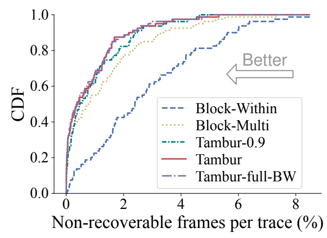
5.3 Sensitivity analysis

There is an inherent trade-off in performance between the non-recoverable frames and bandwidth overhead metrics. The ML model for Tambur is trained using a loss function with a weight of 0.999 on avoiding recovery failures and the remaining weight (i.e., 0.001) on saving bandwidth overhead (§4.3). Fig. 11 shows the impact of this parameter on the frame recovery performance of Tambur with the weight set to 0.9 (i.e., Tambur-0.9) and to 0.5. The improvement in non-recoverable frames for the two schemes are respectively 21.9% and 1.7%. The reduction in the bandwidth overhead is respectively 40.3% and 45.2%. By contrast, recall that Tambur leads to a 26.5% improvement in non-recoverable frames and reduces the bandwidth overhead by 35.1%. Reducing the value of the parameter reduces the frequency of recovering frames and increases the reduction in the bandwidth overhead. Videoconferencing service operators can use these weights as a knob to prioritize one metric over another.

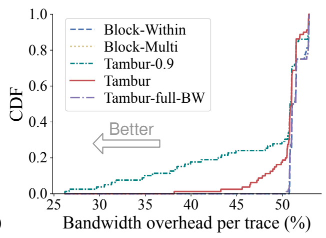
5.4 Online evaluation

Next, we establish Tambur's potential to improve the QoE. To facilitate an easy comparison with the offline evaluation, we show the frequency of non-recoverable losses and the bandwidth overhead (as in §5.2) in Fig. 12. On average, Tambur reduces the number of non-recoverable frames by 69% compared to Block-Within and 34% compared to Block-Multi. Tambur-0.9 reduces the number of non-recoverable frames by 65% compared to Block-Within and 26% compared to Block-Multi despite Block-Multi's much higher latency (Fig. 13b). The results differ slightly from the offline evaluation at the lower percentiles because of setting the parameters of the channel based on average loss statistics over all the traces. This significantly reduced the frequency of calls with low loss rates where any coding scheme suffices to recover nearly all frames (i.e., sophisticated FEC schemes are unnecessary).

Tambur—which is conservative in risking recovery failures to save bandwidth—reduces the bandwidth overhead by 3%



(a) Non-recoverable frames



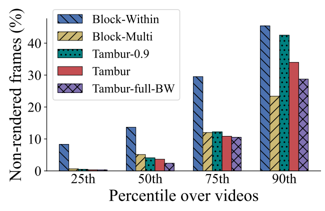
(b) Bandwidth overhead

Figure 12: CDFs for the percent of non-recoverable frames and the bandwidth overhead for the online evaluation.

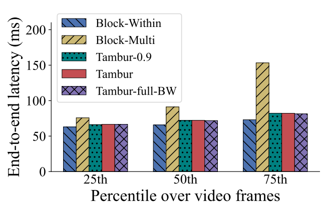
on average of the calls. In contrast, Tambur-0.9 reduces the bandwidth overhead by an average of over 8%. These results reflect both schemes reducing the bandwidth overhead significantly on some calls but only negligibly on many others, which is expected given the loss rates of most calls. Tambur-0.9's bandwidth savings are especially pronounced at the lower percentiles (e.g., 31% at the 10th percentile and 15% at the 20th percentile). Tambur-0.9 provides a win-win by both recovering more frames and saving bandwidth despite the online evaluation reflecting out-of-sample performance for its neural network, which was trained offline over the production traces. The results further validate the trade-off between the bandwidth overhead and recovering frames discussed in §5.3.

Next, we examine the percent of non-rendered frames in Fig. 13a; recall that fewer frames are rendered than recovered due to inter-frame dependencies. Tambur reduces the frequency of failing to render frames compared to Block-Multi and Block-Within by an average of 28% and 73%, respectively. Tambur does worse than Block-Multi at the tail, but this only occurs after all schemes have a failure rate above 23%. Thus, all schemes should employ more redundancy. Tambur-0.9 decreases the frequency of failing to render frames by an average of 70% and 20% compared to Block-Within and Block-Multi, respectively. Tambur-0.9 modestly increases the frequency by 1% at the 75th percentile compared to Block-Multi. Overall, the rate of rendering frames can be improved while simultaneously reducing the bandwidth overhead for most calls. The results are the first to establish the benefits of streaming codes when there are inter-frame dependencies.

Fig. 13b shows that the end-to-end latency is within the upper limit of approximately 150ms for all schemes. Block-Within's latency is slightly lower due to a shorter encode/decode time and always recovering rendered frames using the parity of the same frame (see Fig. 15 and Fig. 16 in Appendix D); Tambur decodes 87% of frames without extra frames versus 88% for Block-Within, so the extra latency from the waiting for extra frames should really be compared to Block-Within failing to decode at all. We argue that Tambur's small cost (e.g., an extra 1.7ms to encode and 3.4ms to decode at the median) is worthwhile due to substantial improvements across the remaining QoE metrics. We also note that our implementation of Tambur's streaming code is not yet optimized



(a) Frequency of non-rendered frames



(b) Latency of rendered frames

Figure 13: Tambur renders significantly more frames than Block-Multi and with lower latency. Tambur's modestly higher latency than Block-Within is more than offset by the improvement in rendering frames.[12]

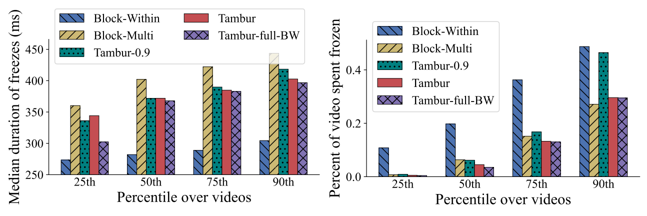
for fast encoding/decoding; hence, we believe it can be significantly faster. Our goal is to establish that Tambur's streaming code is practical enough for videoconferencing applications.

Recall from Fig. 1 that Tambur reduces the frequency of freezes by 78% and 26% compared to Block-Within and Block-Multi, respectively, and Tambur-0.9 reduces the frequency of freezes by 75% and 17% compared to the two respective baselines. Fig. 14a shows that Tambur and Tambur-0.9 each reduce the median duration of freezes compared to Block-Multi by 30ms on average. Tambur and Tambur-0.9 each have a 90ms longer median duration of freezes than theBlock-Within because Block-Within has over 300% more freezes than Tambur does. Many of the extra freezes are short, reducing Block-Within's median value to below Tambur's.

Tambur-0.9 reduces the cumulative duration of freezes by an average of 69% compared to Block-Within. The cumulative duration of freezes is 17% lower for Block-Multi than for Tambur-0.9 despite Tambur-0.9 having on average 11% shorter median durations of freezes and 17% fewer freezes. While the combined effect of the frequency and duration of freezes on the QoE for Block-Multi and Tambur-0.9 are similar, recall that Tambur-0.9 also improves the bandwidth overhead and renders more frames for most traces. As such, we expect Tambur-0.9 to provide an overall higher QoE. Tambur has an average of 77% and 28% shorter cumulative durations of freezes than Block-Within and Block-Multi, respectively, which is a clear win. Tambur, Tambur-0.9, and Tambur-full-BW exhibit higher cumulative durations of freezes at the tail than Block-Multi. We argue that this matters less because the tail QoE is already bad, indicating that all schemes needed more bandwidth overhead. Appendix E explains how this phenomenon is an artifact of the implementation and includes our proposed a solution.

The benefits across QoE metrics of Tambur, Tambur-full-BW, and Tambur-0.9 suggest a markedly improved QoE compared to Block-Within and Block-Multi. Without using ML to reduce the bandwidth overhead, Tambur-full-BW offers a substantial improvement over the two baselines. Tambur and

[12]We omit the 90th percentile since over 10% of frames are not rendered.



(a) Median duration of freezes (b) Percent of video spent frozen

Figure 14: Tambur has a higher median duration of freezes than Block-Within but a significantly smaller cumulative duration of freezes because Tambur has 78% fewer freezes than Block-Within (Fig. 1). Tambur has a lower cumulative and median duration of freezes than Block-Multi.

Tambur-0.9 progressively trade off improvements in loss recovery with bandwidth overhead. Overall, the results illustrate a Pareto frontier of the benefits of streaming codes across the QoE metrics that could be studied further in future work.

6 Related work

FEC for videoconferencing. From the early days of VoIP and Internet-based audio and videoconferencing, FEC has played a key role in recovering lost packets (e.g., [50]). As standards for real-time media and conferencing developed, RTP payload formats for various FEC schemes were defined (e.g., [62]). Later, the FECFRAME working group of the IETF [58] documented traditional FEC schemes such as parity codes [9] and RS codes [57], as well as LDPC [56] and Raptor codes [65]. As the WebRTC project developed based on these standards, it also incorporated the use of FEC to protect media streams [32, 64]. All these codes are block codes. As such, RS codes (i.e., the main baselines against which Tambur was evaluated) have the best loss recovery capabilities of any of them, including fountain [40] and raptor codes [63]. FEC has also been used for rate adaptation. For example, a proposed rate adaptation algorithm for WebRTC, known as FBRA [45], uses extra FEC packets to probe for additional bandwidth, with the benefit that some of the packet losses due to self-induced congestion can be recovered by the FEC.

Streaming codes. In addition to the prior work discussed in §2 and §4, streaming codes have also been studied under various other theoretical models, such as multiplexing with two different decoding delays [4] and multiple burst losses [38]. However, these settings are not directly relevant to our focus on videoconferencing applications.

Alternatives to FEC. Prior work has explored avoiding lossy paths using overlay networks (e.g., Via [34] and J-QoS [31]). While these can be effective in some circumstances, there are two drawbacks to relying only on such approaches. Firstly, these assume that a suitable alternative path *exists* (i.e., that the lossy portion of the path is on a transit network that can be avoided, rather than on the user's home network or last-mile to the ISP, and that there is available interconnectivity with the provider's overlay network); in the current era of hybrid work, enterprises cannot completely eliminate loss through traditional QoS approaches. Secondly, when overlay networks are a feasible solution, there needs to be a careful analysis of when to apply this approach since there is a high financial cost to relaying traffic and fixed capacity on provider networks. Another alternative to FEC is using retransmission to recover losses (e.g., [30]), provided the end-to-end latency is tolerable. However, when there is both high latency and loss (e.g., in cases of acute congestion), retransmission is not always feasible [3]. Tambur provides a flexible end-to-end solution within the application that adapts to any path and is orthogonal to these other approaches.

7 Conclusion

This work introduces Tambur—a new communication scheme for bandwidth-efficient loss recovery for videoconferencing comprising two components. First, a new streaming code that bridges the gap between theoretical streaming codes and videoconferencing applications, which takes as input any given bandwidth overhead. Second, a learning-based predictive model to set the bandwidth overhead. Tambur simultaneously reduces the frequency of non-recoverable frames and the bandwidth overhead by 26.5% and 35.1%, respectively, in our evaluation over large-scale real-world traces from a commercial videoconferencing application. We also design the first videoconferencing framework for implementing and evaluating FEC schemes. The framework enables easy evaluation of the QoE benefits of new communication schemes by providing a simple interface to incorporate (a) new FEC schemes and (b) new learning-based predictive models. Using the framework, we evaluate Tambur and show improvements in QoE metrics, including 26% fewer freezes and 28% fewer non-rendered frames. The benefits establish streaming codes as a viable solution to recovering lost packets for videoconferencing applications. The results thus also show the promise of streaming codes for other live-streaming applications such as cloud gaming.

Acknowledgments

This work was funded in part by an NSF grant (CCF-1910813). We thank our anonymous reviewers for their comments and Nandita Dukkipati for shepherding this work. We also thank Sasikanth Bendapudi, Sivakumar Ananthakrishnan, Nelson Pinto, Jiannan Zheng, and Ross Cutler from Microsoft for their helpful discussions and supporting the project.

References

[1] libvpx. https://chromium.googlesource.com/webm/libvpx/.

[2] WebRTC. https://webrtc.org/.

[3] A. Badr, A. Khisti, W. Tan, and J. Apostolopoulos. Perfecting protection for interactive multimedia: A survey of forward error correction for low-delay interactive applications. *IEEE Signal Processing Magazine*, 34(2):95–113, March 2017.

[4] A. Badr, D. Lui, A. Khisti, W. Tan, X. Zhu, and J. Apostolopoulos. Multiplexed coding for multiple streams with different decoding delays. *IEEE Transactions on Information Theory*, 64(6):4365–4378, June 2018.

[5] A. Badr, P. Patil, A. Khisti, W. Tan, and J. Apostolopoulos. Layered constructions for low-delay streaming codes. *IEEE Transactions on Information Theory*, 63(1):111–141, Jan 2017.

[6] Ahmed Badr, Ashish Khisti, Wai-tian Tan, Xiaoqing Zhu, and John Apostolopoulos. FEC for VoIP using dual-delay streaming codes. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[7] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 339–350, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Dziugas Baltrunas, Ahmed Elmokashfi, Amund Kvalbein, and Özgü Alay. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 225–233. IEEE, 2016.

[9] Ali C. Begen. RTP Payload Format for 1-D Interleaved Parity Forward Error Correction (FEC). RFC 6015, October 2010.

[10] Henrik Boström, Harald Alvestrand, and Varun Singh. Provisional identifiers for WebRTC's statistics API unofficial draft. Draft of a potential specification, W3C, July 2022. https://w3c.github.io/webrtc-provisional-stats/#RTCVideoReceiverStats-dict.

[11] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[12] Ramon Caceres, Alan Clark, and Timur Friedman. RTP Control Protocol Extended Reports (RTCP XR). RFC 3611, November 2003.

[13] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the Google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[14] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. Can you see me now? A measurement study of Zoom, Webex, and Meet. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 216–228, 2021.

[15] Federal Communications Commission. Measuring Broadband America, 2021. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eleventh-report (Last accessed: 2022-02-02).

[16] Mauro Conti, Simone Milani, Ehsan Nowroozi, and Gabriele Orazi. Do not deceive your employer with a virtual background: A video conferencing manipulation-detection system. *CoRR*, abs/2106.15130, 2021.

[17] Ross Cutler, Yasaman Hosseinkashi, Jamie Pool, Senja Filipi, Robert Aichner, Yuan Tu, and Johannes Gehrke. Meeting effectiveness and inclusiveness in remote collaboration. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW1), apr 2021.

[18] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 43–56, 2007.

[19] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 362–373, New York, NY, USA, 2011. Association for Computing Machinery.

[20] E. Domanovitz, S. L. Fong, and A. Khisti. An explicit rate-optimal streaming code for channels with burst and arbitrary erasures. In *2019 IEEE Information Theory Workshop (ITW)*, pages 1–5, 2019.

[21] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2017.

[22] D. Dudzicz, S. L. Fong, and A. Khisti. An explicit construction of optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Communications*, 68(1):12–25, 2020.

[23] E. O. Elliott. Estimates of Error Rates for Codes on Burst-Noise Channels. *Bell System Technical Journal*, 42(5):1977–1997, September 1963.

[24] Martin Ellis. *Understanding the performance of Internet video over residential networks*. PhD thesis, University of Glasgow, 2012.

[25] Salma Shukry Emara, Silas Fong, Baochun Li, Ashish Khisti, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. Low-latency network-adaptive error control for interactive streaming. *IEEE Transactions on Multimedia*, pages 1–1, 2021.

[26] S. L. Fong, A. Khisti, B. Li, W. Tan, X. Zhu, and J. Apostolopoulos. Optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Information Theory*, 65(7):4274–4292, July 2019.

[27] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, Renton, WA, April 2018. USENIX Association.

[28] Boni García, Micael Gallego, Francisco Gortázar, and Antonia Bertolino. Understanding and estimating quality of experience in WebRTC applications. *Computing*, 101(11):1585–1607, 2019.

[29] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40–54, 2011.

[30] Dongsu Han, Ashok Anand, Aditya Akella, and Srinivasan Seshan. RPT: Re-architecting loss protection for content-aware networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pages 71–84, 2012.

[31] Osama Haq, Cody Doucette, John W Byers, and Fahad R. Dogar. Judicious QoS using cloud overlays. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 371–385, 2020.

[32] S. Holmer, M. Shemer, and M. Paniconi. Handling packet loss in WebRTC. In *2013 IEEE International Conference on Image Processing*, pages 1860–1864, 2013.

[33] International Telecommunication Union. ITU-T G. 1010: End-User Multimedia Qos Categories. *G SERIES: Transmission Systems and Media, Digital System and Networks-Multimedia Quality of Service and Performance Generic and User-Related Aspects*, 2001.

[34] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A. Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. VIA: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 286–299, 2016.

[35] M. N. Krishnan and P. V. Kumar. Rate-optimal streaming codes for channels with burst and isolated erasures. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 1809–1813, June 2018.

[36] M. N. Krishnan, D. Shukla, and P. V. Kumar. A quadratic field-size rate-optimal streaming code for channels with burst and random erasures. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 852–856, 2019.

[37] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, page 211–224, New York, NY, USA, 2012. Association for Computing Machinery.

[38] Z. Li, A. Khisti, and B. Girod. Correcting erasure bursts with minimum decoding delay. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 33–39, Nov 2011.

[39] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. GSO-Simulcast: Global stream orchestration in simulcast video conferencing systems. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 826–839, New York, NY, USA, 2022. Association for Computing Machinery.

[40] David J. C. MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

[41] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 229–244, New York, NY, USA, 2021. Association for Computing Machinery.

[42] E. Martinian and C.-E. W. Sundberg. Burst erasure correction codes with low decoding delay. *IEEE Transactions on Information Theory*, 50(10):2494–2502, Oct 2004.

[43] E. Martinian and M. Trott. Delay-optimal burst erasure code construction. In *2007 IEEE International Symposium on Information Theory*, pages 1006–1010, June 2007.

[44] Anush Krishna Moorthy, Lark Kwon Choi, Alan Conrad Bovik, and Gustavo de Veciana. Video quality assessment on mobile devices: Subjective, behavioral and objective studies. *IEEE Journal of Selected Topics in Signal Processing*, 6(6):652–671, 2012.

[45] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion Control using FEC for Conversational Multimedia Communication. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 191–202, 2014.

[46] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, July 2015. USENIX Association.

[47] Ehsan Nowroozi, Ali Dehghantanha, Reza M Parizi, and Kim-Kwang Raymond Choo. A survey of machine learning techniques in adversarial image forensics. *Computers & Security*, page 102092, 2020.

[48] P. Orosz, T. Skopkó, Z. Nagy, P. Varga, and L. Gyimóthi. A case study on correlating video QoS and QoE. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–5, 2014.

[49] Kohong Park and Walter Willinger. *Sele-Similar network traffic and performance evaluation*. Wiley & Son, 2000.

[50] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 12(5):40–48, 1998.

[51] James S. Plank, Ethan L. Miller, Kevin M. Greenan, Benjamin A. Arnold, John A. Burnum, Adam W. Disney, and Allen C. McBride. GF-Complete: A comprehensive open source library for galois field arithmetic version 1.02, 2014.

[52] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[53] Yining Qi and Mingyuan Dai. The effect of frame freezing and frame skipping on video quality. In *2006 International Conference on Intelligent Information Hiding and Multimedia*, pages 423–426, 2006.

[54] Ramya Raghavendra and Elizabeth M. Belding. Characterizing high-bandwidth real-time video traffic in residential broadband networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 597–602. IEEE, 2010.

[55] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[56] Vincent Roca, Mathieu Cunche, and Jerome Lacan. Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME. RFC 6816, December 2012.

[57] Vincent Roca, Mathieu Cunche, Jerome Lacan, Amine Bouabdallah, and Kazuhisa Matsuzono. Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME. RFC 6865, February 2013.

[58] Vincent Roca, Mark Watson, and Ali C. Begen. Forward Error Correction (FEC) Framework. RFC 6363, October 2011.

[59] Michael Rudow and K. V. Rashmi. Learning-augmented streaming codes are approximately optimal for variable-size messages. In *2022 IEEE International Symposium on Information Theory (ISIT)*, pages 474–479, 2022.

[60] Michael Rudow and K. V. Rashmi. Streaming codes for variable-size messages. *IEEE Transactions on Information Theory*, 68(9):5823–5849, 2022.

[61] Michael Rudow and K.V. Rashmi. Online versus offline rate in streaming codes for variable-size messages. *IEEE Transactions on Information Theory*, pages 1–1, 2023.

[62] Henning Schulzrinne and Jonathan Rosenberg. An RTP Payload Format for Generic Forward Error Correction. RFC 2733, December 1999.

[63] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

[64] Justin Uberti. WebRTC Forward Error Correction Requirements. RFC 8854, January 2021.

[65] Mark Watson, Thomas Stockhammer, and Mike Luby. Raptor Forward Error Correction (FEC) Schemes for FECFRAME. RFC 6681, August 2012.

[66] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and

Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[67] Mo Zanaty, Varun Singh, Ali C. Begen, and Giridhar Mandyam. RTP Payload Format for Flexible Forward Error Correction (FEC). RFC 8627, July 2019.

[68] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. SENSEI: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 303–320. USENIX Association, April 2021.

Appendices

A Recovering a burst with Tambur's streaming code

Consider a burst of length b starting in frames i and delay constraint τ. Suppose all frames before the burst have been decoded. First, the received symbols of $P[i],\ldots,P[i+b-1]$ as well as $P[i+b],\ldots,P[i+\tau]$ are used to decode the lost symbols of $V[i],\ldots,V[i+b-1]$. Second, each $U[j]$ for $j \in \{i,\ldots,i+b\}$ is decoded using $P[j+\tau]$. In both steps, decoding follows from solving a system of linear equations.

B Tambur's streaming code's flow network

The graph of the flow network at a high level represents each $P[i]$ that may be used in decoding with a node with an edge into nodes corresponding to each of $U[i]$, $U[i-\tau], V[i],\ldots,V[i-\tau]$, where one unit of flow represents decoding one symbol. The flow network is small (i.e., at most $(5\tau+3)$ vertices and $(2\tau^2+11\tau+5)$ edges for $\tau = 3$). Therefore, the time to solve it is negligible compared to solving the system of linear equations.

C Parameters of the GE channel

To set the parameters of the GE channel for the offline evaluation, we first identify settings that match several aggregate statistics of the production traces as follows. The probability of transitioning from the bad state to the good state (respectively, vice versa) is the mean over traces of one divided by the mean length of bursts (respectively, guard spaces) in frames. The probability of loss in the bad state equals the mean over traces of the multi-frame burstiness. The probability of loss in the good state is then set so that the expected loss rate matches the mean loss rate over traces given the other three parameters. To ensure our results hold for varying network conditions, we then draw the values for each of the four parameters uniformly at random from intervals around these values (rounded to increments of 0.05) as follows. The probability of transitioning from the good state to the bad state and vice versa are distributed as Uniform(0, 0.05) and Uniform(.75, .9), respectively. The probability of loss in the good and bad states are distributed as Uniform(0, 0.05) and Uniform(0.05, 1), respectively.[13]

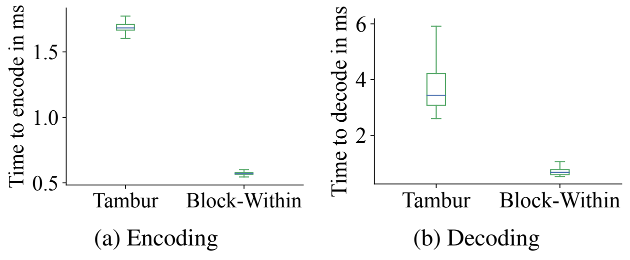


Figure 15: The encoding and decoding times are modest.

D Encoding and decoding overheads

We compare the encoding and decoding time for Tambur with that of Block-Within, which is the fastest of all the baselines (Fig. 15). As seen in Fig. 15, the time to encode and decode is comparable to Block-Within and is only a small fraction of the end-to-end latency budget of 150 ms. The median times for encoding are 1.7ms and .6ms for Tambur and Block-Within, respectively, whereas decoding takes 3.4ms and .7ms for Tambur and Block-Within, respectively. Because Tambur operates over multiple frames of varying sizes, encoding and decoding times are slightly longer and more variable. Our implementation of Tambur requires a fixed amount of memory of approximately 575 KB during encoding and decoding.

But times for encoding and decoding are just a small component of the end-to-end latency. The 50ms one-way delay and the number of extra frames used in decoding (see Fig. 16) have more pronounced effects. Recall that each additional frame used in adds approximately 33 ms to the end-to-end latency, so using fewer extra frames is faster. Tambur does not decode within the same frame only 1% more frequently than Block-Within, which cannot use extra frames in decoding. Tambur uses extra frames to decode only 8% of the time. Block-Multi decodes 24%, 23%, 23%, and 23% of frames with $0,1,2$, and 3 extra frames, respectively. Each extra frame adds $\approx$ 33 ms to the end-to-end latency.

[13]The results were similar when we varied the ranges.

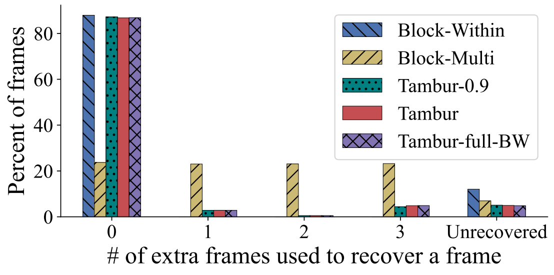


Figure 16: Tambur recovers nearly as many frames as Block-Within using no extra frames and also recovers more overall. Block-Multi recovers an approximately equal number of frames using 0, 1, 2, and 3 extra frames.

E Tail duration of freezes

Recall from Fig. 14b that Tambur, Tambur-0.9, and Tambur-full-BW have higher tail durations of freezes than Block-Multi. The reason for the poor performance is threefold. First, Tambur, Tambur-0.9, and Tambur-full-BW fail to render more frames at the tail, as was discussed in §5.2. Second, the sender generates a keyframe (often ending a freeze) once it learns of recovery failures. Because Block-Within can only recover a frame using the parity packets within the same frame, a keyframe is requested 3 frames sooner (i.e., $\approx$ 100 ms faster) than when Tambur (or Tambur-0.9) is used. Many of the 78% of freezes under the Block-Within where Tambur does not freeze are therefore short and shift the entire distribution of cumulative duration of freezes for Block-Within, including the tail; if we added 0ms freezes for Tambur (or Tambur-0.9) for these instances, their distributions would likewise shift. Third, encoding across multiple frames can make it harder to recover a keyframe triggered by a freeze of several lost frames. This phenomenon does not impact Block-Within and affects Block-Multi less than any of Tambur, Tambur-0.9, and Tambur-full-BW (e.g., does not affect on Block-Multi whenever the keyframe is in the first position within the block of $(\tau+1) = 4$ frames). The phenomenon also contributes to a difference in the frequency of recovered frames (Fig. 12a) and rendered frames (Fig. 13a). There is a natural solution that is outside of the scope of this work. When the sender triggers a new keyframe due to a loss, it should stop taking linear combinations of frames from before the new keyframe. Doing so will strictly (a) increase the frequency of displaying frames and (b) decrease the mean and median duration of freezes. It will benefit Tambur, Tambur-0.9, and Tambur-full-BW the most, but it will also improve Block-Multi to a lesser extent.

F Analysis of recovering bursts

Next, we evaluate Tambur's capabilities for recovering bursts of packets across multiple frames; to do so fairly, we must fix the bandwidth overhead, so "Tambur" refers to Tambur-full-

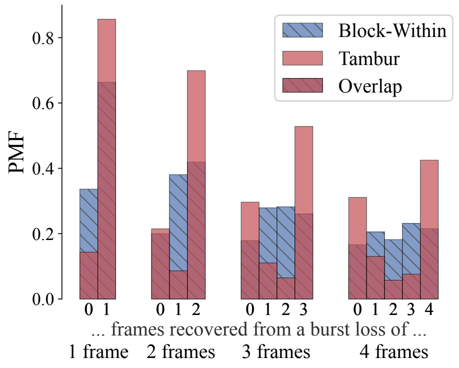


Figure 17: Given the same bandwidth budget as Block-Within, Tambur is more likely to recover all or zero frames from a burst loss over production traces.

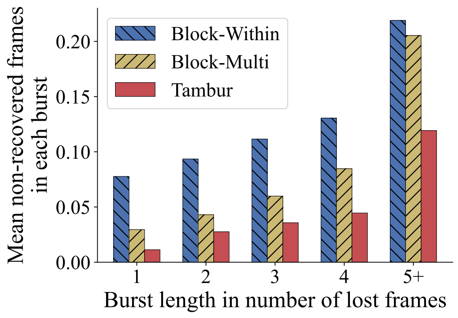


Figure 18: Given the same bandwidth budget as Block-Within/Block-Multi, Tambur provides greater improvement for longer bursts over an emulated network.

BW for the remainder of §F. Fig. 17 shows the distribution of the number of packets recovered for each burst length (in frames) for the offline evaluation. In Fig. 17, the distribution of the number of packets recovered for each burst length (in frames) is shown. Bursts encompassing 2, 3, and 4 frames constitute 23%, 7%, and 3.3% of all lossy events, respectively. For these losses, Tambur recovers all lossy frames 66.8%, 103%, and 97.3% more frequently than Block-Within. For the longer (less frequent) bursts of lengths 3 and 4, when the bandwidth overhead is insufficient, Tambur fails to recover any frames 65.9% and 87% more frequently than the Block-Within. This follows from the Block-Within being more likely to recover some (but not all) of the frames when there is insufficient bandwidth overhead to recover all losses. In contrast, when the bandwidth overhead is insufficient to recover a burst in its entirety, streaming codes are likely to fail to recover all of the frames. However, note that the overall performance of Tambur is still better than the Block-Within: Tambur recovers 21.8%, 12.4%, and 2.3% more frames than the Block-Within for bursts of 2,3, and 4 frames, respectively. Tambur also outperforms Block-Within in recovering losses limited to a single frame, as parity packets sent with later frames can be used in recovery. In short, Tambur performs significantly better for bursts across up to 3 frames than Block-Within and offers more modest gains for bursts across 4 frames.

We also evaluate Tambur's effectiveness at recovering bursts in the online evaluation. Because the loss of a single packet of a frame means that the frame is "lost" under our definition of a burst, longer bursts usually only involve being in the bad state for one, two, or sometimes three frames. We consider the mean number of frames recovered among a burst encompassing 1, 2, 3, 4, or greater than 4 frames in Fig. 18. Tambur reduces the frequency of non-recoverable frames by 70.5%, 68.0%, and 65.8% compared to Block-Within over bursts of 2, 3, and 4 frames respectively. Tambur reduces the frequency of non-recoverable frames by 35.8%, 40.3%, and 47.4% compared to Block-Multi over bursts of 2, 3, and 4, respectively.