GestureFlow: Streaming Gestures to an Audience

Yuan Feng, Zimu Liu, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
{*yfeng, zimu, bli*}*@eecg.toronto.edu*

***Abstract*—Multi-touch mobile devices (*e.g.* iPhone and iPad) and motion-sensing game controllers (*e.g.* Kinect for Xbox 360) share one common feature: users interact with computing devices in non-conventional *gesture-intensive* ways, be they multi-touch gestures on the iPad or body motion gestures with the Kinect. As a new way to interact with computing devices, gestures have been proven to be intuitive and natural, with very minimal learning curve. They can be used in applications beyond games, such as those that allow the creation of artistic and musical content in a collaborative fashion. In order for multiple users to collaborate or compete in real time, however, such gestures need to be streamed in multiple broadcast sessions with an "all-to-all" broadcast nature, with each session corresponding to one of users as a source of a gesture stream. These streams of gestures typically incur low yet bursty bit rates, but have unique requirements with respect to delay and loss. In this paper, we present the design of *GestureFlow*, a gesture broadcast protocol designed specifically for concurrent gesture streams in multiple broadcast sessions. We motivate the effectiveness and practicality of using *inter-session network coding*, and address challenges introduced by linear dependence, discovered in our extensive experiments involving a new gesture-intensive iPad application that we developed from scratch.**

I. Introduction

New mobile devices with large multi-touch displays, such as the iPad, have brought revolutionary changes to ways users interact with computers. Instead of traditional input devices such as keyboards, touchpads and mice, *multi-touch gestures* are used as the primary means of interacting with mobile devices. Surprisingly, *body-motion gestures* are evolving to become a new, natural, and effective way for game players to interact with game consoles in a very similar fashion: in Kinect for Xbox 360, a controller-free gaming experience is made possible by using body-motion gestures to play games [1].

The use of gestures to interact with computers is, of course, not limited to playing action-intensive games. Applications that allow users to create artistic or musical content in an interactive and collaborative fashion may also be gesture-intensive: gestures are frequently needed to create and manipulate artistic strokes or musical notes. These content creation applications share a common feature with interactive games: it is desirable to support collaboration (or competition) among multiple participating users or players. As an example, it would certainly be exciting if music composition hobbyists may collaborate in real time to work on a musical piece.

To support such collaboration among multiple users in real time, we propose that *gestures* are streamed in a broadcast fashion from one user to all participating users, in a broadcast streaming session. Streaming gestures themselves, rather than application-specific data, makes it possible to optimize the design and implementation of a *gesture broadcast protocol* that can be reused by any gesture-intensive application that needs to support multi-party collaboration. We believe this is a more elegant and reusable solution to serve the needs of an entire category of gesture-intensive applications. Once received, a gesture stream can be rendered in real time by a live instance of the same application on a receiver. To take such broadcast of gestures a step further, we believe that *multiple* gesture broadcast sessions need to be supported concurrently, so that any participating user can be the source of a gesture stream.

In this paper, we present *GestureFlow*, a new gesture broadcast protocol specifically designed for multiple concurrent broadcast sessions of user gestures. We point out that gesture streams typically incur low yet bursty bit rates, unlike traditional media streams. They do pose unique challenges, as gesture streams need to be received with the lowest possible delay, and packet losses are not tolerable. In the design of *GestureFlow*, we motivate the use of *network coding*, and present a detailed design that takes advantage of *inter-session* network coding to support low latencies across multiple broadcast sessions.

We believe that the most convincing way to validate our design is to use a real-world implementation of *GestureFlow*, combined with a gesture-intensive application that presents a need for multi-party collaborative creation of content. Towards this objective, we have implemented both *GestureFlow* and *MusicScore* — a multi-touch music composition application — using Objective C from scratch on the iPad. *MusicScore* takes full advantage of our implementation of the *GestureFlow* framework to allow composers to enjoy a live collaborative session.

During extensive experiments presented in this paper, we have discovered new challenges in the use of network coding within the *GestureFlow* implementation. It turns out that coded blocks are linearly dependent with one another with an alarmingly high probability, leading to a much higher overhead than what we originally anticipated. We found that it is due to the fact that the coding window size is typically very small, which is required to satisfy stringent delay requirements. We propose to use systematic Reed-Solomon codes on the source to mitigate the overhead due to such linear dependence.

The remainder of this paper is organized as follows. In Sec. II, we describe the objective and system design of *GestureFlow*. Sec. III presents a thorough analysis of measurement results using our implementation of *GestureFlow* and *MusicScore*, with the observation that coded blocks are linearly

dependent with an alarmingly high probability. In Sec. IV, we propose a solution to mitigate such linear dependence among coded blocks, and evaluate its performance. We discuss related work and conclude the paper in Sec. V and Sec. VI, respectively.

II. GestureFlow: Transporting Gesture Streams

Since gestures are generated by users in real time, the corresponding bit rate of a gesture stream is *low*, but may become bursty. Further, since gestures need to be "replayed" on a receiver by the same application so that application states are affected by these gestures, any missing gestures incur a high risk of inducing inconsistent application states. For example, in a music composition application where a double-tap gesture may be used to add a musical note, if the gesture is not received correctly, the "replay" on a receiver will not add the intended note. Finally, since gestures need to be streamed live, it is important to incur the lowest possible *delay* when transporting these gestures as data packets.

It is intuitive to conceive a design where a TCP connection is established between each pair of users, forming a complete graph of overlay. The reliable and in-order delivery of a stream of bytes is guaranteed by TCP, by using a combination of cumulative acknowledgments, checksums, and retransmissions. However, the realistic nature of traffic on the Internet dictates that overlay links based on TCP connections offer a wide range of available bandwidth and delays, and they vary significantly over time as well. Since TCP uses retransmissions to guarantee reliable delivery, delivery delays may escalate with a slightly more congested link, leading to high delay jitters.

In *GestureFlow*, once a receiver starts playing back a gesture stream, such playback should not be paused. Since intervals between multi-touch gesture events should be maintained to be precisely the same as they are originally generated, the receiver should wait for an initial startup delay before playback begins, and such an initial delay should be sufficiently long to cover any anticipated delay jitter during streaming. A longer initial startup delay should be used to mask a higher delay jitter.

To minimize streaming delays yet with guaranteed reliability, we propose to take advantage of the *"all-to-all"* broadcast nature of *GestureFlow*, where every participating node is the *source* (sender) of a broadcast session to all others, and *multiple broadcast sessions* exist concurrently in the complete overlay graph connecting all users. We advocate two approaches to address the challenges of low-delay streaming with guaranteed reliability.

First, to guarantee reliable delivery of all packets, we propose to take advantage of *random network coding*, to stream *coded* packets using UDP flows rather than TCP, and to allow for possible *relay nodes* in each broadcast session to relay packets that they receive after recoding. *Second,* to minimize streaming delays, it is natural to utilize multiple overlay paths between the source to each receiver, instead of a direct connection in the case of using TCP. Each of these paths is either a direct UDP link between the source and the receiver, or a two-hop path that uses the help of a single relay node. The essence of our transport solution in *GestureFlow* is to use network coding as a rateless erasure code for all broadcast sessions to guarantee reliable delivery, tightly coupled with the use of multiple overlay paths (each of one or two hops) to minimize broadcast delays. In *GestureFlow*, a data packet flows conceptually from each source, in a *coded* form that is mixed with other packets, taking its own path with the lowest delay. Fig. 1 illustrates an example to show how packets are transmitted in coded forms and following different paths.

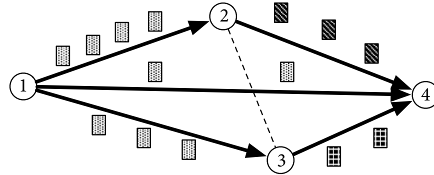

Fig. 1. Streaming coded blocks along multiple paths. Data packets from Node 1 are being transmitted to Node 4 in coded form, either using a direct link, or relayed by Node 2 and Node 3 after being recoded with their own data packets.

A. Coding Gesture Broadcast Sessions

Random network coding has been well established in recent research literature [2], [3], and has been shown to maximize throughput in multicast sessions. With random network coding, k data blocks $\mathbf{b} = [b_1, b_2, \ldots, b_k]^T$, each with s bytes, are to be transmitted from the source to multiple receivers in a network topology. The source transmits coded blocks, each coded block x_j is computed as a linear combination of original data blocks $x_j = \sum_{i=1}^{k} c_{ji} \cdot b_i$ in a finite field (typically $GF(2^8)$), where the set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jk}]$ is randomly chosen. A relay node performs similar random linear combinations on received coded blocks with random coefficients, in order to produce *recoded* blocks to be relayed to a receiver. Coding coefficients related to original blocks b_i are transmitted together with a coded block. A receiver decodes as soon as it has received k linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_k]^T$, either from the source or from a relay. It first forms a $k \times k$ coefficient matrix $\mathbf{C}$, in which each row corresponds to the coefficients of one coded block. It then decodes the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_k]^T$ as $\mathbf{b} = \mathbf{C}^{-1}\mathbf{x}$. Such a decoding process can be performed progressively as coded blocks arrive, using Gauss-Jordan elimination to reduce $\mathbf{C}$ to its reduced row-echelon form. Fig. 2 shows an example of reducing the coefficient matrix to its reduced row-echelon form, after receiving a new coded block.

Decoded (left: b_1; right: b_1, b_2)

$$
\begin{array}{c}
\begin{array}{ccccc} b_1 & b_2 & b_3 & b_4 & b_5 \end{array} \\
\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 1 \end{bmatrix} \\
\begin{bmatrix} 7 & 2 & 3 & 6 & 3 \end{bmatrix}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{ccccc} b_1 & b_2 & b_3 & b_4 & b_5 \end{array} \\
\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 \end{bmatrix}
\end{array}
$$

Fig. 2. Block 2 (b_2) is decoded after receiving the third coded block.

In *GestureFlow*, we have designed a custom-tailored protocol to utilize random network coding in all of the concurrent broadcast sessions, each streaming gesture events from one of the participating nodes. Coupled with a TCP-friendly flow control mechanism such as TFRC, we argue that a coded block is best sent over the UDP transport protocol. The reliable delivery of original data blocks is guaranteed by the use of random network coding. Should a particular coded block be lost, subsequent coded blocks received are equally innovative and useful.

As gesture events are streamed, the size of one gesture event in the stream is very small — less than 1 KB. Without loss of generality, we use touch events and accelerometer values in the iPad as examples. In *GestureFlow*, each original data block contains one touch event if any touch event is generated, along with its preceding accelerometer values. When original blocks of different sizes are coded, the size of the coded blocks is identical to the *largest* original block to be coded. If an original block is smaller when being coded, it is padded with zeros. Since gesture events do not vary substantially in size and the streaming bit rate is very low, the overhead introduced by such padding is not a concern. When each coded block is transmitted as a UDP packet, the identifier of the broadcast session, the starting sequence number of the *earliest* original block that is coded, and all random coefficients are embedded within the same UDP packet so that it is self-contained.

In the theory of random network coding, it is assumed that k data blocks are to be coded, and if more data blocks are being transmitted, they are divided into *groups* of k blocks, and coding is to be performed within each group. If the number of original data blocks k is fixed, it corresponds to a fixed number of gesture events to be coded. We believe, however, that a fixed group size k does not satisfy the needs of streaming gesture events. Due to the inherent bursty traffic of gesture streams, a fixed group size adds to the *delay*, which is critically important in *GestureFlow*. As an example, consider the case where 4 original blocks are to be coded at a sender, yet only 3 gesture events are entered by the user before a long idle period. With a fixed group size, the sender would have to wait for all 4 original blocks to become available before the coding process begins.

To address this challenge, in the *GestureFlow* framework, original data blocks are to be coded within a *sliding window*, referred to as the *coding window*. As a gesture event is generated by the user, it is added to the coding window, as the newest original data block in the window. A maximum size of the coding window, W, is imposed to guarantee successful decoding at receivers, and to manage the coding complexity. The sender performs random network coding on original data blocks within the coding window, and transmits coded blocks to a relay or a receiver as newer original blocks are being added to the coding window. The coding window advances itself by removing the earliest original data block from the window, only when the sender has received acknowledgments from all the receivers in the broadcast session, confirming that the earliest original block has been correctly decoded by all participating nodes.

As a visual illustration of sliding coding windows over time, Fig. 3 shows a typical traffic pattern in *MusicScore* when gesture events are being streamed from a sender to a receiver, and Fig. 4 illustrates how the coding window advances itself, corresponding to the bursts of traffic in *MusicScore*. At time t_1, the coding window R_1 reaches the maximum window size W (4 blocks in this example). As acknowledgments have been received for the first three original blocks, the coding window advances itself by removing them. At time t_2, the coding window R_2 contains 3 blocks. By adopting the sliding window mechanism, during bursty periods when touch events are generated back-to-back, the coding window expands to cover new events, so that they can be received and decoded by receivers in time. During idle times when touch events are scarce, the size of the coding window is naturally reduced as acknowledgments are being received.

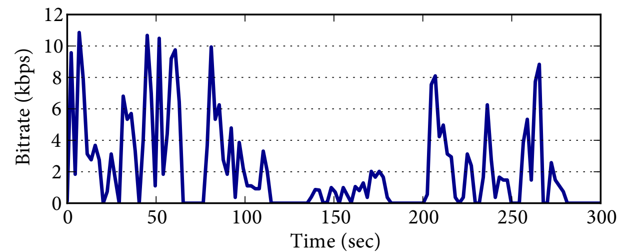

Fig. 3. Bit rates of a typical gesture streaming session in *MusicScore* over time.

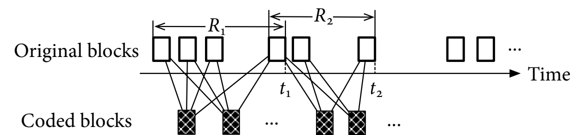

Fig. 4. The coding window advances itself over time.

How should receivers acknowledge the sender of a broadcast session in which an original data block has been correctly decoded? The first intuitive idea is to selectively acknowledge each of the data blocks as soon as they become decoded, even if they are not consecutive to one another. While it is certainly possible for the sender to remove any of the original blocks from the coding window when they are acknowledged by all the receivers, it requires a coded block to carry the sequence numbers of all original blocks that are coded. Since sequence numbers of original blocks are not consecutive, it is no longer feasible to carry only the starting sequence number of the *earliest* original block. In *GestureFlow*, we believe that the additional overhead and complexity are not justified, and propose to use *cumulative* acknowledgments, which are much simpler.

With cumulative acknowledgments of decoded data blocks, a receiver uses Gauss-Jordan elimination to reduce the coefficient matrix of all coded blocks it has received so far to its reduced row-echelon form (RREF), and finds out which block

has just been completely decoded. Instead of acknowledging a newly decoded data block immediately, the receiver sends an acknowledgment for a decoded block only if *all* earlier blocks with smaller sequence numbers have been decoded. As an example shown in Fig. 5, the receiver does not acknowledge b_3 even though it has been decoded after receiving two coded blocks x_1 and x_3. It waits until receiving another coded block x_4, which renders all three data blocks, b_1, b_2, and b_3, decoded.

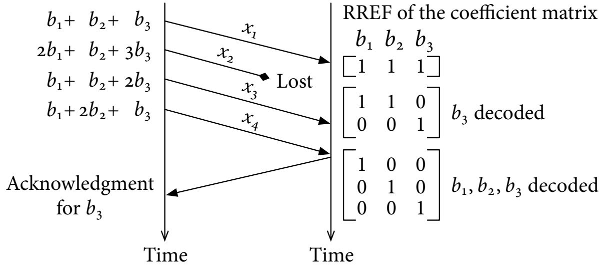

Fig. 5. A receiver sends a cumulative acknowledgment only when all earlier blocks have been decoded.

B. Coding Across Multiple Broadcast Sessions

In the *GestureFlow* design, coded blocks are transmitted to the receiver via multiple paths. They are either sent directly to the receiver as UDP packets (via a *single-hop* path), or relayed by one of the participating nodes (via a *two-hop* path). The motivation for including two-hop paths via relay nodes is our stringent requirement for low streaming delays when gesture events are streamed. In each broadcast session, since all relay nodes are receivers themselves, no additional bandwidth is consumed to take advantage of two-hop paths. If these paths offer lower delays than the direct link, each original data block will arrive at a receiver via a path with the lowest delay, yet in a coded form. Over time, the *delay jitter* is reduced by sending data blocks with random network coding via multiple paths.

When relay nodes are used, what should each relay node do as it receives coded blocks belonging to multiple broadcast sessions, each streaming gesture events from one of the participating nodes? Naturally, a relay node should be allowed to *recode* incoming coded blocks and transmit them to others. Since we only consider using two-hop paths in *GestureFlow* — it would be unlikely that a three-hop path offers a lower delay than a direct one — such recoding is to be performed only *once*: it should only be performed on coded blocks from the source of a broadcast session.

Incoming coded blocks to a relay node, however, belong to different broadcast sessions. Shall we allow for recoding *across* multiple broadcast sessions? This is referred to as *inter-session* network coding in the theoretical literature, but it has not yet been adopted in any practical systems using network coding. If inter-session network coding is performed on a relay node, it mixes coded blocks from multiple broadcast sessions together when it sends outgoing recoded blocks. More specifically, all incoming coded blocks — as well as those from the relay node's own broadcast session in which it is the source — are mixed together and then transmitted to all other participating nodes. Recall the four-node example in Fig. 1, and consider all 4 streaming sessions from 4 users. The relay coding engine with inter-session network coding in Node 2 is shown in Fig. 6.

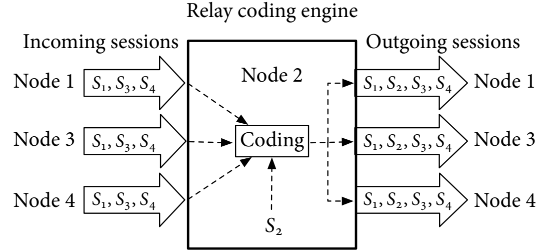

Fig. 6. The relay coding engine with inter-session network coding in Node 2, in the four-node example shown in Figure 1.

In the *GestureFlow* framework, we have made the decision that all relay nodes are to perform network coding across multiple broadcast sessions. The advantages of inter-session network coding are two-fold. *First*, it is *simple* to design and implement on a relay node. Due to the broadcast nature of each session, every participating node is the source of a broadcast session to all others. If a relay node mixes all incoming coded blocks together in its recoding process, there is no longer a need to allocate outgoing bandwidth to multiple concurrent sessions, or to schedule outgoing blocks belonging to different sessions competing for outgoing bandwidth. With inter-session network coding, a relay node only needs to transmit as many coded blocks as the outgoing bandwidth allows, without consideration of the sessions they belong to. *Second,* thanks to low bit rate streams in *GestureFlow*, the number of coded blocks belonging to a broadcast session is small. If recoding is restricted within a session, it is possible that very few blocks are recoded, leading to a higher probability of linear dependence in the decoding process. By mixing coded blocks from multiple sessions together, a relay node increases the number of original blocks in an outgoing coded block.

The only challenge that remains now is: What is the set of coded blocks that is to be recoded on a relay node to produce an outgoing coded block? Of course, if an original data block is already decoded at a downstream receiver of a relay node, it should not be included in its recoding process. In other words, the relay node should also maintain a *sliding* recoding window, and remove original data blocks that are no longer useful to receivers. But how does a relay node know which original block is already decoded at receivers?

The simple answer to this question is: the relay node does not know directly, but the source knows. In the *GestureFlow* design, we have stated that a receiver sends cumulative acknowledgments to the source of a session, confirming the latest original block that has been decoded. With the inclusion of relay nodes and multiple paths, such acknowledgments are sent directly from a receiver to the source of a session, and are not sent to any of the relay nodes. Nevertheless, after the

source of a session advances its coding window by removing its earliest original block, all relay nodes will easily detect such an advance, as the sequence number of the earliest original block is embedded within a coded block.

It only remains to see when a relay node removes a block from its buffer. In *GestureFlow*, a relay node does not mandate a stringent maximum size for its recoding window, which includes received coded blocks in its buffer. As a coded block arrives, it adds its coefficient row to the existing coefficient matrix, and reduces the new matrix to its reduced row-echelon form (RREF). When recoding, it simply recodes all rows in the existing matrix, which is in RREF. Even if an original data block is completely decoded when reducing the coefficient matrix to its RREF, the relay node does not remove the block from its recoding window immediately, since doing so introduces the risk that subsequent original blocks may not be decoded. Instead, the relay node waits until the source of the session advances its coding window, and removes an original block from its recoding window *only* when it is no longer included in the coding window of the source of the session. Since the source only advances its coding window when all receivers have decoded an original block, recoding such a block will no longer benefit any of the receivers. In other words, a relay node removes an original data block from its recoding window, if its sequence number is smaller than the starting sequence number in a newly received coded block from the source.

To illustrate how blocks are removed from the recoding window on a relay node, Fig. 7 shows coefficient rows of coded blocks in Node 4's recoding window, in the context of our four-node example. $b_i^{(j)}$ represents an original data block i in the jth broadcast session. In the figure, we observe that Node 4 has decoded the first original data block of Session 1, 2 and 3 by receiving the first four coded blocks. The last coefficient row corresponds to a newly received coded block. It no longer includes a non-zero coefficient for $b_1^{(1)}$ and $b_1^{(2)}$, which indicates that the coding windows in Node 1 and Node 2 have advanced beyond these original data blocks. As a relay node, Node 4 now removes coefficient rows that correspond to these two blocks, which implies that these blocks are removed from its recoding window, within which it produces outgoing coded blocks in the future.

$b_1^{(1)}$	$b_2^{(1)}$	$b_3^{(1)}$	$b_1^{(2)}$	$b_2^{(2)}$	$b_1^{(3)}$	$b_2^{(3)}$	$b_1^{(4)}$	
1	0	0	0	0	0	0	0	Removed from the recoding window
0	1	5	0	7	0	8	9	
0	0	0	1	0	0	0	0	Removed from the recoding window
0	0	0	0	0	1	0	0	
0	3	6	**0**	4	2	8	1	

Fig. 7. Removing data blocks from the recoding window of a relay node: an example.

III. Experiences with GestureFlow

We dedicate this section to investigations of how *GestureFlow* performs in real-world systems. We implemented *MusicScore*, a collaborative music composition application, from scratch with the iPad Programming SDK. Users interact with *MusicScore* to compose music using only multi-touch gestures. *MusicScore* takes full advantage of the *GestureFlow* framework to stream gesture events among multiple participating users, such that composers can enjoy a live collaborative experience. Both *MusicScore* and the *GestureFlow* framework have been implemented in Objective-C in the Xcode programming environment. Based on the Model-View-Controller (MVC) design pattern, *MusicScore* utilizes many aspects of Cocoa frameworks provided by Apple Inc., including the Core Animation API. Fig. 8 shows a screenshot of a live *MusicScore* composition session on the iPad, in which a highlighted note has just been selected by a user.

Fig. 8. A screenshot of music composition in *MusicScore* on the iPad: a selected note is highlighted in blue.

To minimize the computational load on the iPad, we have included an optimized implementation of random network coding in the *GestureFlow* framework. Our implementation of network coding is able to progressively decode incoming coded blocks using Gauss-Jordan elimination, while taking full advantage of SIMD instructions available in the ARM Cortex A8 architecture, used by CPUs powering both the iPad and the iPhone 3GS. The *GestureFlow* implementation contains over $8,000$ lines of code.

Unless otherwise specified, our experiments are conducted using four nodes in "all-to-all" broadcast sessions, with two iPads connecting to the Internet through Wi-Fi, and two iPhone 3GS devices connecting to the Internet through 3G and EDGE, respectively.

A. *Overall Performance of* GestureFlow

We first present streaming delays achieved in *MusicScore* by using *GestureFlow*. Table I summarizes the measured average round-trip time, $\overline{x}$, and its standard deviation, s, between each pair of connection types by probing every one of them for 100 times over time. In comparison, Table II presents streaming

delays — the time from gesture events being generated at the source to blocks being decoded at the receiver — in *GestureFlow*, and their standard deviations. It is clear from a comparison between these tables that, though for each pair of users the measured streaming delay in *GestureFlow* is slightly higher than the corresponding one-way delay (which corresponds to half of the round-trip time), the standard deviation of streaming delays in *GestureFlow* is much lower than that of one-way delays. As a consequence, the initial startup delay, which is used to mask delay jitters with a buffer, is reduced significantly. In this way, the intervals between gestures can be kept precisely.

TABLE I
ROUND-TRIP TIME MEASUREMENTS (IN MILLISECONDS).

$(\overline{x}, s)$	Wi-Fi 1	Wi-Fi 2	3G	EDGE
Wi-Fi 1	–	$(153, 43)$	$(359, 195)$	$(573, 357)$
Wi-Fi 2	$(153, 43)$	–	$(305, 148)$	$(459, 322)$
3G	$(359, 195)$	$(305, 148)$	–	$(617, 401)$
EDGE	$(573, 357)$	$(459, 322)$	$(617, 401)$	–

TABLE II
STREAMING DELAYS WITH *GestureFlow* (IN MILLISECONDS).

$(\overline{x}, s)$	Wi-Fi 1	Wi-Fi 2	3G	EDGE
Wi-Fi 1	–	$(103, 25)$	$(192, 104)$	$(309, 181)$
Wi-Fi 2	$(89, 20)$	–	$(167, 88)$	$(274, 178)$
3G	$(224, 118)$	$(188, 121)$	–	$(364, 267)$
EDGE	$(347, 209)$	$(301, 194)$	$(398, 282)$	–

Next, we evaluate an important design choice adopted in *GestureFlow*: utilizing multiple overlay paths between the source and every receiver to minimize streaming delays. Fig. 9 shows the average percentage of blocks each kind of receiver received from relay nodes, along with the 95% confidence interval. We observe that for Wi-Fi and 3G users more than 10% of the received blocks are from relay nodes, and for the EDGE user up to 30% of received blocks are from relay nodes. The reason is that an EDGE user usually has higher streaming delays on direct links. As a result, it is more likely for the EDGE user to receive coded blocks from relays that have lower delays, rather than from senders directly.

To investigate the scalability of *GestureFlow*, we further study the correlation between the streaming delay and the number of participating users. Note that all participating nodes are Wi-Fi users in this experiment. As shown in Fig. 10, the average streaming delay varies mildly around 100 ms, as the number of nodes increases. We can even observe a slight decrease in the average streaming delay when the number of nodes is large, *e.g.*, 14 nodes, which is due to an increased number of one-hop relay paths that may have lower streaming delays.

To illustrate the bandwidth overhead in *GestureFlow*, we investigate the difference between the gesture streaming bit rate, which is computed as the average of four broadcast sessions, and the upload bit rate per user, which is defined as the average upload bit rate each user devotes to every broadcast session. As shown in Fig. 11, the gap between these

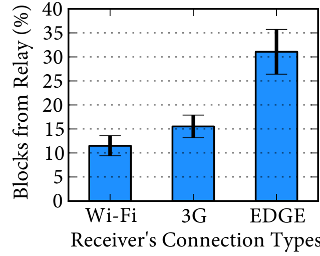

Fig. 9. Percentage of recoded blocks from relay nodes over all received blocks.

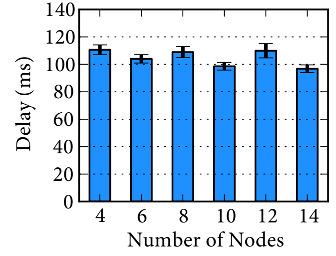

Fig. 10. Delays with different numbers of participating nodes.

two curves gets larger when the streaming bit rate is higher. The reason is that during bursty periods, every user has to contribute more bandwidth to upload coded blocks containing blocks from other sessions, which introduces more bandwidth overhead. Yet, the bandwidth overhead for each user is less than 5 kbps in general, which is reasonable. It is critical to point out that even with overhead considered, the upload bit rate per user is only about 8 kbps on average, which is fairly low in streaming systems. This verifies our design philosophy that bandwidth is not a major concern in *GestureFlow*.

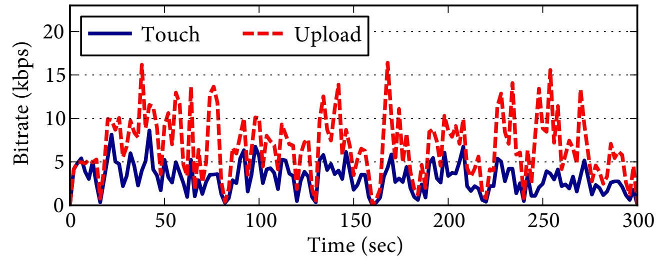

Fig. 11. Bandwidth overhead per user.

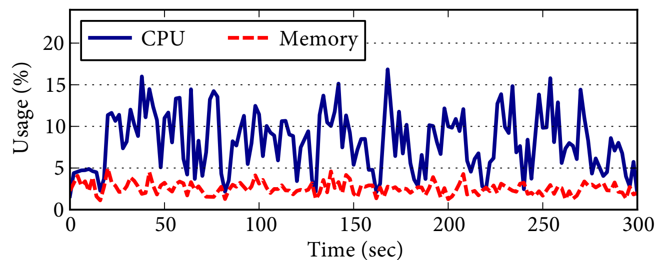

Fig. 12. The CPU load and memory usage of network coding in an iPhone 3GS device.

Beyond overhead, another concern in *GestureFlow* is its CPU load and memory usage by adopting network coding, which are mainly introduced by Gauss-Jordan elimination in the decoding process. We have measured the CPU load and memory usage over time at an iPhone 3GS node, and plot the results in Fig. 12. As indicated, the average CPU usage is 8.4%, with peaks corresponding to bursty bit rates in Fig. 11. The dashed line shows the memory usage over time, which is 2.4% on average. The iPad has an even lower CPU load

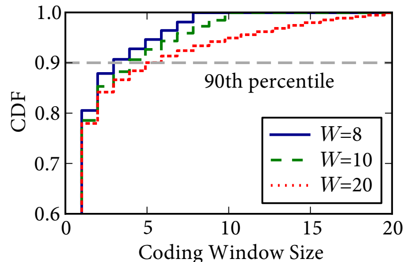

Fig. 13. CDF of the coding window size in the source.

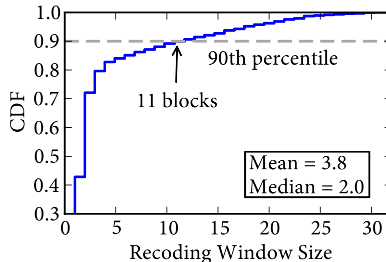

Fig. 14. CDF of the recoding window size in a relay node.

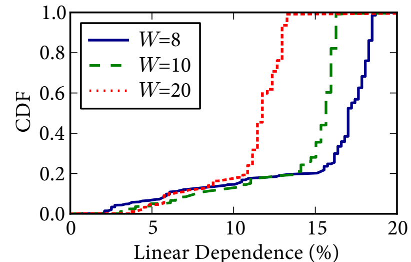

Fig. 15. CDF of linear dependence in different settings of maximum coding window size (W).

as it enjoys an even higher CPU frequency in its Cortex A8 architecture, and the same memory usage as the iPhone 3GS (both have 256 MB of main memory). As such, the CPU load and memory usage of network coding in *GestureFlow* are very acceptable.

B. The Performance of Network Coding in GestureFlow

By applying network coding in *GestureFlow*, it is important to evaluate its performance as well. Fig. 16 shows the relationship between the maximum coding window size W and the streaming delay. We observe that the streaming delay increases when W is getting either smaller or larger, and reaches its minimum when W equals to 8. The underlying reason is that when W is set to be too small, the source needs acknowledgments for almost every block to advance the coding window. Subsequent blocks have to wait a longer time before they can be coded and transmitted, which increases the delay, especially in bursty periods. On the other hand, if W is too large, the received coded blocks always contain coding coefficients for newly coded blocks, which increases the delay in the decoding process.

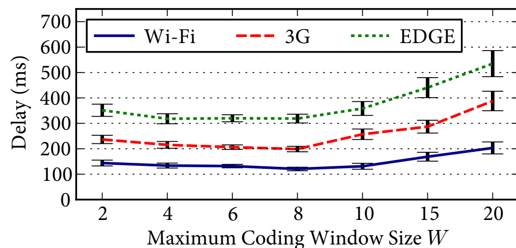

Fig. 16. The average delays along with 95% confidence intervals in different experiment settings.

Having evaluated the maximum coding window size, we now proceed to analyze flexible coding windows in both sources and relay nodes. We plot the CDF of the actual coding window size in the source and the recoding window size in the relay node, which are shown in Fig. 13 and Fig. 14, respectively. From Fig. 13, we can see that 90% of the coding windows have a size of no more than 5 blocks. This indicates that most of the time there is little delay added in both the coding and decoding processes, as blocks do not have to wait too long to be transmitted or relayed. The underlying reason is that *GestureFlow* has very bursty traffic, since users idle for most of the time, which reduces the actual coding window naturally. Similarly, Fig. 14 shows that 90% of the recoding windows have a size of no more than 11 blocks, with an average of around 4 blocks. This indicates that, in general, there is only one block or two from each broadcast session required to be recoded at the relay node, which justifies the use of inter-session network coding. By mixing a limited number of coded blocks from multiple sessions together, recoded blocks generated by relay nodes are useful to downstream receivers with high probability.

It is critical to point out that coded blocks, either from senders or recoded ones from relay nodes, are considered useful only when they are *linearly independent* with each other, or else they are regarded as redundant blocks. We further investigate the ratio of linear dependence among coded blocks with different coding window size W. For a specific data block, its linear dependence is computed as the percentage of linearly dependent blocks over the total number of coded blocks involving the data block. As shown in Fig. 15, we plot the CDF of linear dependence of all data blocks in our experiment. We can see that 90% of coded blocks contain around 15% blocks that are linearly dependent with each other, which is alarmingly high. The linear dependence ratio is even higher when the coding window size is getting smaller, 18% when $W = 8$.

A high percentage of linear dependence among coded blocks implies a large portion of redundant blocks, which unnecessarily consumes bandwidth. Though we emphasize that gesture streams typically incur very low bit rates, they are highly bursty as well. Shown in Fig. 3, the bursty bit rate reaches 10 kbps in a session. The bandwidth consumption due to linearly dependent blocks may escalate with concurrent broadcast sessions.

IV. GestureFlow: Mitigating Linear Dependence

To mitigate the bandwidth overhead due to a high percentage of linearly dependent blocks with small coding window sizes, we propose to apply *systematic Reed-Solomon codes* based on the Vandermonde matrix on the source node in a *GestureFlow* broadcast session.

With *systematic* Reed-Solomon codes, the source sends original blocks first, rather than sending coded blocks from

the onset. These original blocks can be seen as a special case of coded blocks, with coding coefficients as rows in an identity matrix. After sending all original blocks, the source starts to generate and send coded blocks. In order to code k original blocks using an (n, k) systematic Reed-Solomon code over a Galois field F_q, the source is able to generate up to $n-k$ coded blocks, after original blocks are transmitted. In *GestureFlow*, a $(n-k) \times k$ Vandermonde matrix G of the following form is used to generate these coded blocks:

$$G = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & k \\ 1 & 2^2 & 3^2 & \cdots & k^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{n-k-1} & 3^{n-k-1} & \cdots & k^{n-k-1} \end{bmatrix}.$$

Since matrix G is a Vandermonde matrix, it is easy to see that any $k \times k$ submatrix of G has a non-zero determinant and is nonsingular, and as a result every subset of k rows of G is guaranteed to be linearly independent. As such, linear independence among all original and coded blocks transmitted from the source is guaranteed with the use of the Vandermonde matrix to generate coded blocks. These blocks from the source are always innovative once received. An additional benefit is that, sending original blocks eliminates the need of generating coding coefficients at the source of a broadcast session, and mitigates the computational overhead of performing Gauss-Jordan elimination at a receiver.

Compared with random linear nodes, one difference of using systematic Reed-Solomon codes at the source is that Reed-Solomon codes are not *rateless*. With the $(n-k) \times k$ Vandermonde matrix G, a maximum of $n-k$ coded blocks can be sent, in addition to the original blocks. In contrast, with a randomized generation of code vectors, random network coding is able to produce a practically infinite number of coded blocks to ensure successful decoding with any erasure channel.

In practice, though, this is not a serious limitation in *GestureFlow*. We have shown in Sec. III-B that the optimal coding window size, W, is 8, which implies that $k \leq 8$, and the receiver is able to decode successfully as long as k linearly independent blocks — original or coded — are received. Since W is set to be so small, even if a standard size of the Galois field $q = 256$ is used, and Galois field arithmetic is performed on GF(256) during coding, n can still be chosen to be as large as $q - 1 = 255$, which means that the code used is a $(255, k)$ code where $k \leq W$. This is indeed a linear code with a very low rate, and implies that decoding will be successful with high probability. In situations where the packet loss rate is so high that fewer than k linearly independent blocks are received, the session is considered to be terminated.

Other parts of the transport protocol in *GestureFlow* remains the same, in a sense that the cumulative acknowledgments, progressive decoding, one-hop relay, and inter-session network coding are still adopted. Note that relay nodes still use random linear network coding to recode received blocks, so that there are no restrictions imposed on the recoding window size.

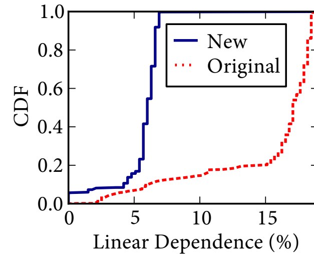

Fig. 17. CDF of linear dependence with systematic Reed-Solomon codes used at the source nodes ($W = 8$).

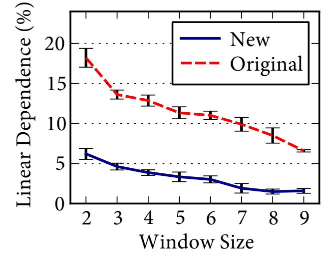

Fig. 18. Linear dependence with different coding window sizes.

The effectiveness of using a systematic Reed-Solomon code at the source in *GestureFlow* is evaluated with additional experiments with *MusicScore*. Fig. 17 shows the comparison of CDFs of linear dependence between the original *GestureFlow* design and the use of systematic Reed-Solomon codes to mitigate linear dependence. The maximum coding window size W is set to be 8. It is clear that the 90th percentile of linear dependence is significantly reduced by using the new design, from 18% to 6%. Since blocks — either original ones or coded using the Vandermonde matrix — from the source are guaranteed to be linearly independent with each other, the linear dependence is caused by the recoding process in relay nodes, which is acceptably low. The ratio of linear dependence with different coding window sizes is explored in Fig. 18. We can see that the ratio of linear dependence is decreasing as the coding window size increases, and the ratio with a systematic Reed-Solomon code used at the source is much smaller than the original *GestureFlow* design.

Since the most critical design objective in *GestureFlow* is to satisfy a stringent delay requirement, we compare streaming delays and their standard deviations from the Wi-Fi 1 User to other three users by using the new and original *GestureFlow* designs, respectively, shown in Table III. We can observe that by using systematic Reed-Solomon codes at the source, average streaming delays through different kinds of connections have all been evidently reduced. Since the redundancy due to linear dependence is mitigated with the use of systematic Reed-Solomon codes at the source, a received block can be used to decode with a higher probability, reducing the decoding delay.

TABLE III
STREAMING DELAYS (IN MILLISECONDS) AT WI-FI 1 USER WITH NEW *GestureFlow* DESIGN.

$(\overline{x}, s)$	Wi-Fi 2	3G	EDGE
New	$(85, 23)$	$(177, 105)$	$(287, 178)$
Original	$(103, 25)$	$(192, 104)$	$(309, 181)$

In summary, our experiments in Sec. III and Sec. IV have evaluated our important design decisions made in *GestureFlow*, with the objective of reducing streaming delays and delay jitters. Our results have confirmed that, by allowing one-hop relays, using systematic Reed-Solomon codes at source

nodes, and using random linear network coding at relay nodes with sliding windows, streaming delays and delay jitters are effectively reduced. We have also shown that *GestureFlow* scales well when the number of participating users increases, while the overhead and computational load introduced by network coding are acceptable.

V. Related Work

With the inception of network coding [4] and random network coding [2], [3] in information theory, the topic has attracted a substantial amount of research attention. Analytical studies [3], [4] have shown that network coding is able to maximize information flow rates in multicast sessions in direct acyclic graphs. In more practical systems, Gkantsidis *et al.* [5], [6] have shown that the use of random network coding in peer-to-peer file sharing systems can reduce the time to download files. Annapureddy *et al.* [7], [8] have evaluated the use of network coding in peer-to-peer on-demand streaming systems, and have shown that network coding helps to achieve good performance with respect to the sustainable playback rate and system throughput. In contrast, the use of network coding in *GestureFlow* is specifically designed for streaming low bit-rate traffic from gesture events, and for ensuring reliable delivery with low delays.

With respect to the design of transport protocols, network coding has been applied to improve the system performance. For example, Chachulski *et al.* [9] have proposed MORE, which improves the unicast throughput in the context of wireless opportunistic routing by using intra-session network coding. CodeCast, presented by Park *et al.* [10], is a network coding based multicast protocol for low-latency multimedia streaming. Sundararajan *et al.* [11] have proposed a modified acknowledgment mechanism to incorporate network coding into TCP, with the objective of providing better support to a unicast session. In their solution, the number of blocks involved in the sender's sliding window is completely controlled by TCP. The receiver acknowledges the degree of freedom of the coefficient matrix of coded blocks received so far. Network coding is used in a separate underlying layer as a rateless erasure code, and is decoupled from window-based flow control in TCP.

In comparison, *GestureFlow* is remarkably different. It is designed specifically for multiple broadcast sessions, each involving a stream of gesture events. Acknowledgments in *GestureFlow* serve the purpose of indicating to the source when an original block has been correctly decoded by all receivers in a broadcast session. Rather than leaving the control of the sliding window at the source to TCP, the *GestureFlow* design dictates very specific rules about how the coding window advances itself, and about what the maximum window size is. Receivers are more conservative in that they only acknowledge blocks that are completely decoded, and in a cumulative fashion. Finally, there have been no discussions in the literature — including [11] since it does not involve multiple sessions — about why and how inter-session network coding should be implemented in practice.

VI. Concluding Remarks

We are firm believers that gestures represent a new paradigm for users to interact with mobile devices, and that social and collaborative aspects of gesture-intensive applications will usher in an era of *streaming* gesture events live, so that applications do not need to design and implement custom-tailored solutions. We are intrigued by the very low yet bursty bit rates when streaming gesture events over the Internet, as shown in a real-world application — *MusicScore* — that we have developed from scratch to compose music collaboratively on mobile devices such as the iPad. Such low streaming bit rates, coupled with the need for guaranteed reliability, low streaming delays, and multiple concurrent broadcast sessions when multiple users are involved, have brought us brand new but very practical challenges that need to be addressed with a new transport solution.

While designing the *GestureFlow* framework, we have tried a number of alternative designs, governed by the principles of *simplicity* and *practicality*. This paper presents our design of using random network coding with multiple one-hop or two-hop paths, allowing for recoding across multiple concurrent sessions. We intend to present not only *how* our design in *GestureFlow* works, but also *why* we have chosen such a design. The use of network coding has simplified our design and implementation, making them more practical. In closing, we are in the hope that this paper only represents the first step towards a mature framework that facilitates the streaming of gestures, so that users interact with one another in a simple and transparent fashion to create or consume multimedia content, wherever they may be around the world.

References

[1] [Online]. Available: http://www.xbox.com/en-CA/kinect/

[2] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. Allerton Conference on Communications, Control and Computing*, 2003, pp. 40–49.

[3] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting," in *Proc. IEEE Internationl Symposium on Information Theory (ISIT '03)*, 2003, p. 442.

[4] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[5] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive View of a Live Network Coding P2P System," in *Proc. Internet Measurement Conference (IMC '06)*, 2006, pp. 177–188.

[6] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. IEEE INFOCOM '05*, vol. 4, 2005, pp. 2235–2245.

[7] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena, and P. Rodriguez, "Is High-Quality VoD Feasible Using P2P Swarming?" in *Proc. Internationl WWW Conference '07*, 2007, pp. 903–912.

[8] ——, "Exploring VoD in P2P Swarming Systems," in *Proc. IEEE INFOCOM '07*, pp. 2571–2575.

[9] S. Chachulski, M. Jennings, S. Katti, and D. Katabi, "Trading Structure for Randomness in Wireless Opportunistic Routing," in *Proc. ACM SIGCOMM*, 2007, pp. 169–180.

[10] J.-S. Park, M. Gerla, D. Lun, Y. Yi, and M. Medard, "CodeCast: A Network-Coding-Based Ad Hoc Multicast Protocol," *IEEE Wireless Communications*, vol. 13, no. 5, pp. 76–81, 2006.

[11] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, "Network Coding Meets TCP," in *Proc. IEEE INFOCOM '09*, 2009, pp. 280–288.