GestureFlow: QoE-Aware Streaming of Multi-Touch Gestures in Interactive Multimedia Applications

Yuan Feng, *Student Member, IEEE,* Zimu Liu, *Student Member, IEEE,* and Baochun Li, *Senior Member, IEEE*

***Abstract*—With the proliferation of multi-touch mobile devices, such as smartphones and tablets, users interact with devices in non-conventional gesture-intensive ways. As a new way to interact with mobile devices, gestures have been proven to be intuitive and natural with a minimal learning curve, and can be used in interactive multimedia applications. In order for multiple users to collaborate in an interactive manner, we propose that gestures can be streamed in multiple broadcast sessions, with each session corresponding to one of the users as the source of a gesture stream. During the interactive session, the Quality of Experience (QoE) of mobile users hinges upon delays from when gestures are entered by the source to when they are recognized by each of the receivers, which we refer to as *gesture recognizing delays*. In this paper, we present the design of *GestureFlow*, a gesture broadcast protocol designed specifically for concurrent gesture streams in multiple broadcast sessions, such that the gesture recognizing delay in each session is minimized. We motivate the effectiveness and practicality of using *inter-session network coding*, and address challenges introduced by the linear dependence of coded packets. We evaluate our protocol design using an extensive array of real-world experiments on mobile devices, involving a new gesture-intensive interactive multimedia application, called *MusicScore*, that we developed from scratch.**

***Index Terms*—Multi-Touch Streaming, Inter-session Network Coding, Mobile Framework.**

I. Introduction

NEW mobile devices with multi-touch displays have brought revolutionary changes to ways users interact with wireless devices, with *multi-touch gestures* used as the primary means of interaction. In particular, interactive multimedia applications on mobile devices have made it possible to use gestures intensively to create and consume artistic or musical content in an interactive and collaborative fashion, since gestures are frequently needed to create and manipulate artistic strokes or musical notes. With such media authoring applications, it is desirable, if feasible, to support collaboration among multiple participating users. 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 session. Streaming gestures themselves, rather than application-specific data, has made it possible to optimize the design and implementation of a *gesture broadcast protocol* that can be reused by any mobile multimedia application that needs to support multi-party collaboration. Clearly, it is a more elegant and reusable solution to serve the needs of an entire category of gesture-intensive media applications. Once received, gestures can be recognized and rendered in real time by a live instance of the same application on a receiver. To take such broadcast of gestures a step further, *multiple* gesture broadcast sessions need to be supported concurrently, so that any participating user can be the source of a gesture stream.

With such gesture streaming broadcast sessions in place, a high-quality user experience within an interactive application hinges upon an important Quality of Experience (QoE) metric: the time it takes for a gesture to be recognized at each of the receivers, starting from the time it is recognized at the source of the session. Referred to as the *gesture recognizing delay*, such a delay is an application-layer QoE metric that directly affects the user-perceived quality of an application session.

In this paper, we present *GestureFlow*, a new QoE-aware gesture streaming protocol specifically designed for multiple concurrent broadcast sessions of gestures, with the objective of minimizing *gesture recognizing delays* in these broadcast sessions. Since only a subset of raw touch events can be recognized as multi-touch gestures, the source of a broadcast session needs to send *raw* touch events, which will be recognized at each of the receivers. Each recognized gesture will consist of a number of network-layer packets, all of which need to be received for the gesture to be correctly recognized. Unlike traditional media streams, gesture streams typically incur low yet bursty bit rates, but packet losses are not tolerable since each lost packet will severely affect the accuracy of the gesture recognizer on a receiver.

In order to support multiple broadcast sessions while minimizing gesture recognizing delays and guaranteeing reliable packet delivery, we present a detailed design that uses *inter-session network coding*, and addresses a number of open challenges in our design that have not been discussed in the literature: what the best size of the coding window should be, how the coding window should be advanced, and how packets from different sessions can be coded together.

To validate our design, we have developed a real-world implementation of *GestureFlow*, as well as an interactive music composition application, called *MusicScore*, 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

Manuscript received on August 8, 2011; revised on January 23, 2012.

Y. Feng, Z. Liu, and B. Li are with the Department of Electrical and Computer Engineering, University of Toronto, Canada (e-mail: {yfeng, zimu, bli}@eecg.toronto.edu).

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, and show that the revised QoE-aware gesture streaming protocol has met our needs with the smallest possible gesture recognizing delay.

The remainder of this paper is organized as follows. Sec. II discusses the motivation and challenges of QoE-aware gesture streaming. In Sec. III, we describe our detailed system design in *GestureFlow*, using inter-session network coding. In Sec. IV, we present a thorough analysis of measurement results using our implementation of *GestureFlow* and *MusicScore*, and observe that coded blocks are linearly dependent with an alarmingly high probability. In Sec. V, we propose our solution to mitigate such linear dependence among coded blocks, and evaluate its performance. We discuss related work and conclude the paper in Sec. VI and Sec. VII, respectively.

II. Gesture Streaming: Motivation and QoE

Multi-touch allows users to interact with user interface elements directly with their fingers using *gestures*, and has been proven to be the most intuitive interface for a wide variety of applications. These gestures can be as simple as a one-finger tap, or as complicated as a three-finger swipe. As a running example and experimental testbed, we have designed and implemented an iPad application for music composition using multi-touch gestures, called *MusicScore* and shown in Fig. 1, from scratch. *MusicScore* allows a user to create musical notes with double taps, to change the pitch of notes by dragging them vertically, and to select a group of notes by dragging a rectangle around them.

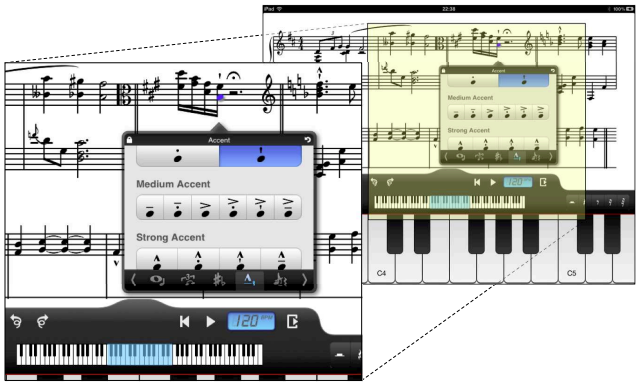


Fig. 1. A screenshot of running *MusicScore* on the iPad.

A. Streaming Multi-touch Gestures

In *MusicScore*, there are situations in which a teacher gives her student a tutorial on music composition when she is traveling; or composition hobbyists collaborate to compose a piece of music without being physically together. These examples have shown a clear need to facilitate spontaneous sharing of user experience among multiple users, which substantially improves the utility of multi-touch applications. In a nutshell, rather than streaming application-specific data, we propose that multi-touch gestures are streamed instead, regardless of what the application may be. With the same application running on multi-touch devices belonging to all participating users, the streamed gestures can be precisely rendered on a receiving device, as if they are entered *live* by the local user.

By streaming multi-touch gestures, we immediately gain a number of important benefits. In contrast to the design of customized state exchange protocols for specific collaborative media applications, such as musical symbols in a score and artistic objects in a canvas, it would be much more generic to stream multi-touch gestures as representatives of user interactions. By handling the replay of streamed gestures as user input, application states are updated correctly at the receiving device. This implies that gesture streaming can serve as an underlying framework that supports any media application that desires multi-party collaboration. Furthermore, streaming multi-touch gestures makes it easier for multiple users to interact with the same set of application states at the same time. In *MusicScore*, this implies that multiple users are able to compose the same piece of music together, by composing different voices or musical instruments in the piece.

Since all participating users are able to affect the state of the application, it is a must that everyone can see the exact changes made by other users. While some multi-touch gestures can easily be replayed after being streamed to a different user, such as adding a note in *MusicScore*, other gestures only change the views of the local user, such as zooming, and do not affect the state of the application. If participating users have different views, it will be impossible to show all of them on one display. We solve this problem by adopting a *"picture in picture"* design: a user's own gestures interact with the native view using the full-screen display, and the views of participating users are displayed in their respective overlay windows. In the example shown in Fig. 2, both Alice and Bob are able to work on their preferred views, and to observe the view of the other party at the same time.

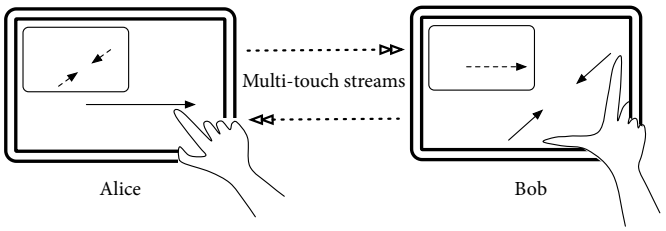


Fig. 2. A "picture in picture" design as multi-touch gestures are streamed between two users, who are collaborating to compose the same piece of music.

B. Quality of Experience

Our ultimate design objective is to design a reusable framework, called *GestureFlow*, from the ground up to stream multi-touch gestures to multiple participating users, with the best possible Quality of Experience achieved. The framework uses a shared set of well-designed presentation and transport mechanisms to support a variety of interactive multimedia applications, including *MusicScore*. There are a number of challenges when designing the *GestureFlow* framework.

First, multi-touch gesture streams have a *very low*, yet *bursty*, bit rate. In multi-touch applications, it is usually the case that users interact with their devices frequently for a while and then stay idle most of the time. For example, a music composer touches the display in *MusicScore* to add or remove notes only when she is inspired. Fig. 3 shows the bit rates of a typical gesture broadcast session in *MusicScore* over time. We can observe that the peak bit rate reaches 11 kbps, while the average bit rate is no more than 3 kbps.

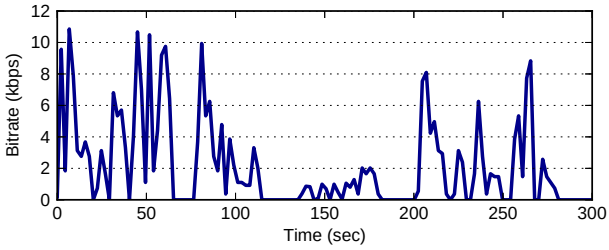


Fig. 3. Bit rates of a typical gesture broadcast session in *MusicScore* over time.

Second, multi-touch gestures need to be streamed in an *in-order*, *lossless* and *error-free* fashion, as any lost or erroneously transmitted gesture to any of the participating users makes it difficult to precisely render and reconstruct application states at the receiver. This is different from typical media streams, where a loss or an error is considered an inconvenience that degrades playback quality, but not a catastrophic event.

Third, gesture streaming has a stringent delay requirement. Media applications that need the support from a gesture streaming framework are interactive in nature, and demand the smallest possible *gesture recognizing delay*, from when gestures are recognized by the source, to when they are eventually received and recognized by each of the receivers.

Finally, once the replay of streamed gestures has started at a receiver, the interval between the replay of two consecutive gestures has to be kept identical to the difference between their original timestamps when they are generated at the sender. Otherwise, rendered states of an application may be different from the original. This implies that each gesture has to be *recognized* at the receiver *on time*, *i.e.*, before its scheduled *replaying* time, despite the fact that each gesture may be received with different end-to-end delays over the Internet, and as a result experience different gesture recognizing delays. Similar to live media streaming, an *initial startup delay* — with a corresponding application buffer at the receiver — can be used to mask varying gesture recognizing delays.

Using our "Bob and Alice" example, we illustrate delays in the session from Alice to Bob in Fig. 4. Alice's gestures are received by Bob with end-to-end delays $\tau_1, \tau_2, \tau_3, \ldots$, and recognized by Bob's application with gesture recognizing delays $v_1, v_2, v_3, \ldots$. Though a gesture may be recognized by Bob's application, its replay may be delayed by an initial startup delay δ, such that the intervals between gestures, $\Delta t_1, \Delta t_2, \ldots$, are kept precisely the same during replay. In order to make sure all gestures are recognized on time for replaying, the initial startup delay δ has to be no shorter than the longest gesture recognizing delay, $v_{\max}$. Therefore, to achieve the best possible Quality of Experience with a short δ, gesture recognizing delays need to be minimized.

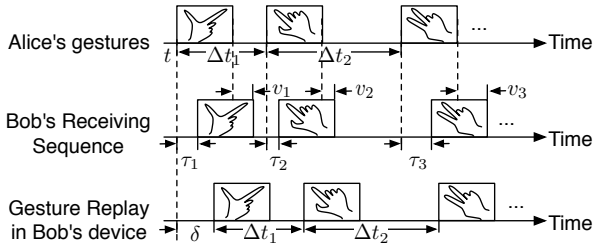


Fig. 4. The replay of gestures from Alice to Bob, using an initial startup delay δ to mask varying end-to-end delays τ_i over the Internet, as well as gesture recognizing delays v_i in the application.

With these unique characteristics, the design of the *GestureFlow* framework is more challenging than conventional media streaming systems. It needs to be designed so that a bursty and low bit-rate stream from each user can be transmitted to all participating users in a reliable and timely fashion, in a set of broadcast communication sessions.

C. Presenting Multi-touch Gestures

We are now ready to present a detailed design of our *GestureFlow* framework. The first natural question is how gestures should be presented and packetized, in preparation for streaming to multiple participating users.

In multi-touch applications, *gesture recognizers* are instances that analyze a *raw stream* of touch objects in a sequence, and determine the intention of users based on properties of each gesture. It analyzes the number of touches and the number of taps from the raw stream, and compares them with the required ones stored in the recognizer to make a decision. Table I shows descriptions and examples in *MusicScore* for a collection of useful gestures.

What information should be streamed for a precise playback at a receiver? There are two alternatives. The first is to use a *raw stream*, represented by a successive sequence of *touch events* (*e.g.*, finger-down, finger-up, or location update of touch), and the second is to stream a sequence of gestures recognized by gesture recognizers. Intuitively, one may think streaming recognized gestures would be sufficient for mobile applications. Unfortunately, *GestureFlow* needs to use the first alternative, and to stream raw touch events rather than recognized gestures. This is because interactive media applications typically include a mixture of raw touch events and recognized gestures. Take a drawing application as an example, while scaling or moving an object in the canvas can be done by gestures, artistic drawing requires a raw stream of touch events to track every movement of fingers. Table II summarizes examples of multi-touch operations in typical interactive applications, which require both raw touch events and recognized gestures.

As soon as the raw stream is received, gestures can be recognized by the same application on the receiver, as illustrated in Fig. 5. As a gesture is essentially a sequence of raw touch events, before the last touch event arrives at the receiver

TABLE I
EXAMPLES OF MULTI-TOUCH GESTURES IN *GestureFlow*.

Gesture type	*Description*	*Example in MusicScore*	*Information needed to replay*
Tap	Tap a view with one or more fingers, possibly multiple times	Select a note or chord	The number of fingers used, the number of taps, and the location in a given view
Swipe	One or more fingers moving towards a direction for a distance	Scroll up or down	The location of the first touch, the direction of the swipe, its velocity and the distance
Touch and hold	Touch with one or more fingers and hold for a short period of time	Trigger a pop-up menu to change the note duration or to add accidentals	The location in a given view

TABLE II
EXAMPLES OF MULTI-TOUCH OPERATIONS IN TYPICAL INTERACTIVE APPLICATIONS.

Interactive Application	*Brushes* (Artistic drawing)	*MusicScore* (Music composition)
Touch Events	Draw a curve with varying speed	Free play on the virtual piano keyboard
Recognized Gestures	Move a layer in the canvas using panning	Create a note using double-tapping

and leads to a successfully recognized gesture, all preceding touch events have already being streamed to the receiver. In other words, the recognizer in the receiving application has been *progressively* receiving "partial" information about this gesture; once the last raw touch event is received, the gesture is immediately recognized. In contrast, if a gesture is streamed when it is fully recognized by the sender, the receiver has to wait for the transmission of an entire gesture, which is typically larger than a touch event, *i.e.*, the delay of recognizing a gesture in the receiving application will be higher.

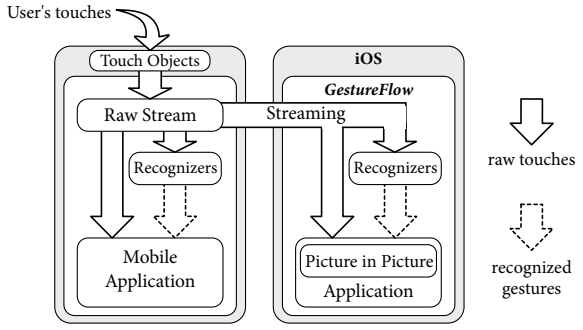


Fig. 5. Streaming a raw stream of touch objects.

To packetize and transmit a raw stream of touch objects (Fig. 5), we propose to use a simple binary format, due to the fact that there are only four types of events involved (touch-begin, touch-move, touch-end and touch-cancel). With a raw stream, touch objects are continuously generated as a user interacts with her device, and transmitted with a compact form of presentation so that the bit rate is minimized. Each of these touch events should be accompanied by its *sequence number* and *timestamp*. At a receiver, sequence numbers are used to detect out-of-order delivery and losses of the event stream, and timestamps help to replay them with precise time intervals as they are originally generated: any time interval Δt_i can be computed as the difference between the timestamp of the i^{th} event and that of the $(i-1)^{\text{th}}$.

III. TRANSPORTING GESTURE STREAMS

When transporting gestures, we wish to achieve the best possible Quality of Experience, in that gesture recognizing delays are to be as short as possible. It is intuitive to conceive a design where a TCP connection is established between each pair of users, forming a complete graph of overlay. Although TCP guarantees the reliable and in-order delivery of a stream of bytes, the realistic nature of traffic on the Internet dictates that overlay links based on TCP connections exhibit a wide range of delays, and vary significantly over time as well. Further, since TCP uses retransmissions to guarantee reliable delivery, delays may escalate with a slightly more congested link, leading to high delay jitters.

To minimize end-to-end delays of delivering gestures to receivers with guaranteed reliability, we present two approaches by taking advantage of the "all-to-all" broadcast nature of *GestureFlow*, where every participating node is the source node of a broadcast session to all others, and multiple broadcast sessions exist concurrently in the complete overlay graph connecting all users. *First*, we propose to use *random network coding*, which streams coded blocks using UDP flows rather than TCP, and allows possible relaying nodes to relay blocks that they receive after recoding. *Second*, instead of direct connections in the case of using TCP, multiple paths between the source and each receiver are used to minimize the end-to-end delays of delivering gestures. Since all relaying nodes are receivers themselves, no additional bandwidth is consumed to take advantage of relay paths. The essence of our transport solution 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 paths to minimize end-to-end delays.

A. GestureFlow: Design Overview

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. The reliable delivery of original data blocks is guaranteed with the erasure correction nature of random network coding. Should a particular coded block be lost, subsequent coded blocks received are equally innovative and useful.

Data blocks flow conceptually from each source in a *coded* form that is mixed with other blocks, via multiple single-hop or two-hop paths to each destination, with each two-hop path using one participating node as a relay. With multiple paths, original data blocks will arrive at a receiver via the path

with the lowest delay, yet in a coded form. Fig. 6 illustrates an example to show how blocks from Node 1's session are transmitted in coded forms and following different paths.

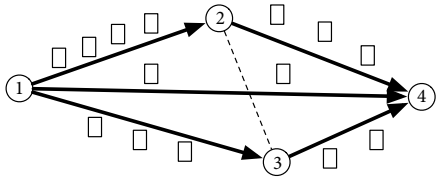


Fig. 6. Streaming coded blocks from Node 1 to all other participating nodes along multiple paths. Data blocks from Node 1 are being transmitted to Node 4 in coded form, either using a direct link, or relayed by Node 2 and Node 3 after being recoded with their own data blocks.

When participating nodes are used to relay data blocks, they are also producing their own original blocks. What should each node do with these data blocks belonging to multiple broadcast sessions? Since a node is capable of recoding all coded blocks it has received before transmitting them to others, shall we allow for recoding *across* multiple broadcast sessions? If we do, a participating node would then serve the dual role of being a source node and a relaying node. Referred to as *inter-session* network coding in the theoretical literature, it has not yet been adopted in any practical systems using network coding.

In the *GestureFlow* framework, we have made the decision that all nodes are to perform network coding across multiple broadcast sessions. If each node is allowed to mix all incoming blocks with original blocks produced by itself, 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, every node only needs to transmit as many coded blocks as the outgoing bandwidth allows, without considering the sessions they belong to.

In the four-node example shown in Fig. 6, if we consider all 4 broadcast sessions from 4 users concurrently, the inter-session network coding engine in node 2 is shown in Fig. 7. As we can see, node 2 produces coded blocks covering all 4 sessions, each of which carries the necessary information other nodes require to decode, such as the sequence numbers of original blocks, all random coefficients, and the coded payload, so that it is self-contained.

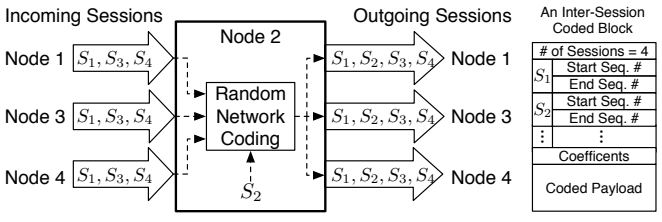


Fig. 7. The inter-session network coding engine in node 2, in the context of the four-node topology shown in Figure 6, with 4 "all-to-all" broadcast sessions considered.

B. GestureFlow: Protocol Design

Now that we have presented an overview of *GestureFlow*, we are ready to discuss more details in our protocol design.

Basics of random network coding. Random network coding has been well established in recent research literature [1], [2], and has been shown to maximize throughput in multicast sessions. With random network coding, k original 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. A source of a broadcast session generates coded blocks $x_j = \sum_{i=1}^{k} c_{ji} \cdot b_i$ as a linear combination of original data blocks in a finite field (typically $GF(2^8)$), where the set of coding coefficients $\mathbf{c}_j = [c_{j1}, c_{j2}, \cdots, c_{jk}]$ is randomly chosen. A relaying node is able to perform similar random linear combinations on received coded blocks with random coefficients. Coding coefficients related to original blocks b_i are transmitted together with a coded block.

A receiver is able to decode all k data blocks when 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 even be performed progressively as coded blocks arrive, using Gauss-Jordan elimination to reduce $\mathbf{C}$ to its reduced row-echelon form (RREF).

Although inter-session network coding is conceptually simple to perform, its real-world design and implementation have brought a number of challenges to the spotlight. In what follows, we illustrate our design choices as we address these challenges.

The size of an original data block. When determining what the value of s — the size of an original data block — should be, we have discovered in our experiments that the size of one gesture event in the stream is very small: less than 512 bytes. In *GestureFlow*, each original data block contains one touch event if any touch event is produced. When original blocks of different sizes are coded, they are padded with zeros to the size of 512 bytes. 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.

Cumulative acknowledgments. How should receivers acknowledge the source node 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 source node to remove any of the original blocks from the coding window when it is 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. Since the additional overhead and complexity may not be justified, we 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 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. 8, 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.

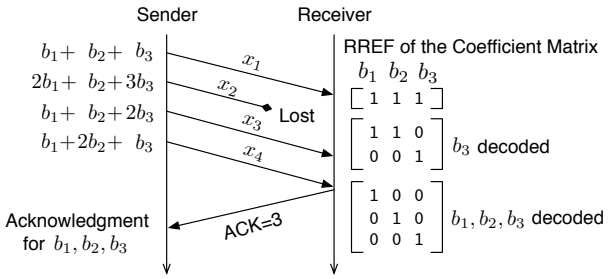


Fig. 8. A receiver sends a cumulative acknowledgment only when all earlier blocks have been decoded.

Basics of the coding window at a source node with no relaying. 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 k is fixed, it corresponds to a fixed number of blocks to be coded. However, a fixed group size k may negatively affect the QoE metric in gesture streaming: due to inherently bursty traffic when gestures are streamed, a fixed group size may increase the gesture recognizing delays. As an example, consider the case where 4 blocks are to be coded at a source node, yet only 3 are received or produced, followed by a long idle period. With a fixed group size, the source node would have to wait for all 4 blocks to become available.

To address this challenge, blocks are to be coded within a *sliding window* in *GestureFlow*, referred to as the *coding window*. To explain the basic idea of a coding window, let us first consider the simplified scenario where a node does not relay coded blocks from other broadcast sessions, *i.e.*, only original blocks are coded by the source node of a broadcast session. In this case, as a new original block containing a new gesture event is produced, it is added to the coding window at the source node. A maximum size of the coding window, W, is imposed to guarantee successful decoding at receivers, and it corresponds to the maximum number of original blocks that can be coded to produce an outgoing coded block. The source node performs random network coding on original blocks within the coding window, and sends coded blocks to all the receivers as newer blocks are being added to the coding window. The coding window advances itself by removing the earliest data block from the window when the source node has received acknowledgments from all the receivers in the broadcast session.

Fig. 9 shows the basic idea of the coding window at a source node. At time t_1, the coding window grows to 3 as block 5 enters, and then reaches the maximum coding window size W (4 blocks in this example) at time t_2. Note that even though blocks 7 and 8 have already been produced containing new multi-touch events (and buffered), they are not added to the coding window since it has already reached its maximum size. After a few coded blocks are received, the receiver acknowledges that blocks $3-5$ have been successfully decoded. At time t_3, the coding window at the source node advances itself by removing acknowledged blocks, and then blocks 7 and 8 enter the coding window. By adopting the sliding window mechanism, during bursty periods when touch events are produced 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.

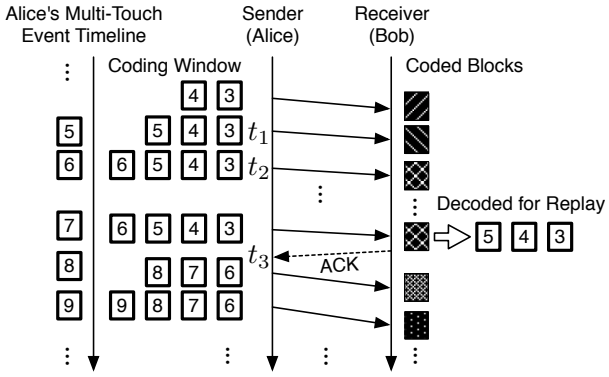


Fig. 9. The coding window at a source node advances itself over time.

Shrinking the maximum size of the coding window on demand. Since raw touch events are streamed directly and a gesture consists of multiple raw touch events, with a fixed maximum size for the coding window, blocks containing information to recover one gesture may be split into separate coding windows, incurring longer gesture recognizing delays at the receivers. Longer gesture recognizing delays can be reduced if we can send out a coded block *immediately* when the recognizer on the source node recognizes a gesture.

In our design, the source node will always "shrink" the maximum size of the coding window to the current block whenever a gesture is recognized, so that no other blocks can enter the current coding window. By doing so, the source node will not need to hold itself back and wait for new original blocks after a gesture is already recognized; it is also easier for receivers to receive enough coded blocks to recover the gesture since fewer blocks are contained in a coded block. Fig. 10 shows an illustration of our maximum coding window size adjustment mechanism with the previous example in Fig. 9. When a gesture is recognized by the recognizer at time t_4, the source node will adjust its maximum size of the coding window to 2 blocks, and send out these coded blocks immediately. When the source node receives acknowledgments confirming that blocks belonging to this gesture are successfully decoded by all receivers, the source node advances its sliding window, and the maximum window size is then restored to its original value (4 blocks).

The coding window with inter-session network coding. Unfortunately, the design of the coding window in *GestureFlow*

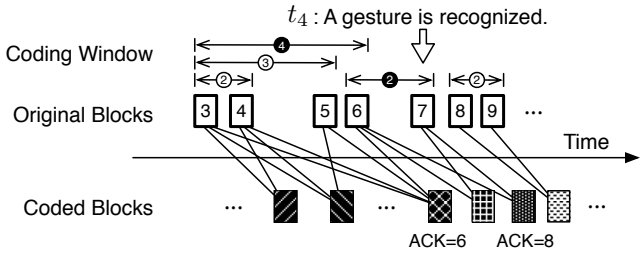


Fig. 10. The source node adjusts its maximum coding window size W when a gesture is recognized by its gesture recognizer. The number in the circle indicates the size of the coding window; and the dark circle indicates that the maximum coding window size has been reached.

becomes more complicated with inter-session network coding, where a source node of a broadcast session also serves as a relaying node for other sessions. The initial complexity comes from the computation of the coding window size. Even though a source node also serves as a relay and mixes incoming coded blocks from other sessions, these incoming coded blocks should not affect the computation of the coding window size. In other words, the coding window size should still be the number of original blocks that the source node itself has produced, and the source node will not include new original blocks in its coding window if it has reached its maximum size.

As we mix blocks from multiple broadcast sessions, what is the set of blocks that is to be coded on a node for an outgoing block to be produced? Of course, if an original data block is already decoded at a downstream receiver, it should not be included in the recoding process. As a result, with inter-session network coding, the coding window at each node should *selectively* remove original data blocks within other broadcast sessions, if they are no longer useful to all the receivers. But how does a node in its role as a relay know which original block is already decoded at receivers?

The short answer to this question is: the relaying node does not know directly, but the source node knows, since receivers sends *cumulative acknowledgments* to the source node of a session, acknowledging the latest original block that has been decoded. Of course, these acknowledgments are sent directly from a receiver to the source node of a session, and are not sent to any of the relaying nodes. Nevertheless, after the source of a session advances its coding window by removing its earliest original block, all relaying 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 node removes a block from its coding window. As a coded block arrives or as an original block is produced, it adds its coefficient row to the existing coefficient matrix, and reduces the new matrix to its RREF. After eliminating original data blocks from its own broadcast session that are acknowledged by all receivers, it simply recodes all rows in the existing matrix, even if an original data block from another broadcast session is completely decoded when reducing the coefficient matrix to its RREF. The node does not remove the block from its coding window immediately, since doing so introduces the risk that subsequent original blocks may not be decoded. Instead, a node, in its role as a relay, waits until the source node of a broadcast session advances its own coding window. The relaying node removes an original block from its coding window *only* when its sequence number is smaller than the starting sequence number in a newly received coded block from the source node of a session. Since the source node 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.

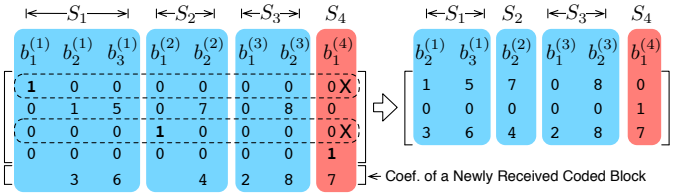


Fig. 11. Removing data blocks from the coding window of a node in its role as a relay (node 4).

To illustrate the design of the coding window with inter-session network coding, in the context of our four-node example given in Fig. 7, Fig. 11 shows the coefficient matrix at node 4. $b_i^{(j)}$ represents an original data block i in the j^{th} broadcast session, S_j. In the left-side matrix in RREF, we observe that node 4 has decoded $b_1^{(1)}$ and $b_1^{(2)}$, from S_1 and S_2 respectively, by receiving the first three coded blocks, and the fourth coefficient row corresponds to $b_1^{(4)}$ from node 4 itself for inter-session network coding. In a newly received coded block (the coefficients of this block are in the last row), any information related to $b_1^{(1)}$ and $b_1^{(2)}$ is no longer included, which indicates that the coding windows at node 1 and node 2 have advanced beyond these original data blocks. In its role as a relay, node 4 now removes coefficient rows that correspond to these two blocks (circled by dashed rectangles) from its coding window, within which it produces outgoing coded blocks in the future.

In summary, Algorithm 1 describes the *GestureFlow* protocol using inter-session network coding.

IV. 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, consisting over $59,000$ lines of code. 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. Fig. 12 shows a scenario of a live *MusicScore* composition session on the iPad, in which collaboration is achieved using *GestureFlow*.

To minimize the computational load on the iPad, we have included an optimized implementation of random network coding in the *GestureFlow* framework. Our implementation

Algorithm 1 *GestureFlow* running on the source node of session S_j.

Event: Received a multi-touch event
1: Encapsulate the new multi-touch event into an original block, $b_i^{(j)}$, with proper zero-padding.
2: **if** the maximum size of the coding window has not been reached **then**
3: Include $b_i^{(j)}$ into the coding window
4: Increment the size of the coding window
5: **end if**
Event: Received an ACK from a receiver in the session S_j
6: Compute the smallest sequence number r from all ACKs received so far from receivers.
7: Advance the coding window by removing all original blocks before $b_r^{(j)}$ (inclusive).
8: Include more buffered original blocks into the coding window, if any, until the maximum size of the coding window has been reached.
9: Recompute the size of the coding window based on the number of original blocks included.
Event: Received a coded block
10: Add the coded block to the coding window.
11: Reduce the coefficient matrix (corresponding to blocks in the coding window) to its RREF using Gauss-Jordan elimination.
12: **if** $b_p^{(q)} (q \neq j)$ and earlier blocks can be decoded **and** $b_{p+1}^{(q)}$ cannot be decoded **then**
13: Decode blocks till $b_p^{(q)}$ (inclusive)
14: Send ACK containing p to the source node of S_q
15: **end if**
16: **if** $b_i^{(q)} (i \leq p, q \neq j)$ is not included in the received coded block **then**
17: Removing blocks associated with $b_i^{(q)}$ from the coding window
18: **end if**
Event: The network is ready for a block to be transmitted
19: Produce and transmit a linear combination of all blocks in the coding window with randomly generated coefficients.

Fig. 12. *MusicScore* in action: two users are collaboratively composing a musical piece with support from the *GestureFlow* framework.

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 v7 architecture, used by CPUs powering the iPad (all generations) and the iPhone (including 3GS, 4 and 4S). The *GestureFlow* implementation itself contains over $8,000$ lines of code.

A. Performance Analysis

As our primary QoE metric, we first present measurement results with respect to the gesture recognizing delays. In each run of our experiments, we measure the gesture recognizing delay in a collaborative music composition session between a pair of iPads running *MusicScore*, and the corresponding CDF curve is derived from multiple runs of our experiments. Our experiments are performed in both Wi-Fi networks and 3G cellular networks to better capture the performance of *GestureFlow*. Given a type of Internet connectivity (Wi-Fi or 3G), two iPads are connected to the Internet via two different ISPs, reflecting a more dynamic network condition. We have also implemented a traditional TCP-based streaming protocol, named *TCP Relay*, as a baseline for our comparisons. For fairness, *TCP Relay* also transmits data blocks through both direct TCP link and two-hop relay paths to minimize both end-to-end delays and delay jitters.

As shown in Fig. 13 and Fig. 14, when *GestureFlow* is used, averages of gesture recognizing delays are 102.6 msec and 253.3 msec for Wi-Fi and 3G users, respectively. In contrast, *TCP Relay* suffers from much longer gesture recognizing delays: 183.6 msec and 485.1 msec on average, for Wi-Fi and 3G users, respectively. The shorter delays achieved by *GestureFlow* can be attributed to both the adoption of inter-session network coding and a sliding coding window, specifically designed for gesture streaming. Besides, since devices in cellular networks (3G or EDGE) cannot directly connect with each other via TCP, due to NAT restrictions, they have to connect to the same set of relays with publicly accessible IP/port (*e.g.*, dedicated relay servers for users in cellular networks) for data exchange. This results in longer end-to-end delays between 3G users, as shown in Fig. 14. In contrast, UDP-based *GestureFlow* can easily achieve NAT traversal in cellular networks, and as a result achieve shorter end-to-end delays.

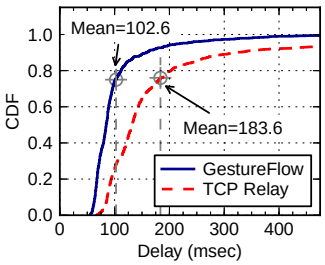


Fig. 13. CDF of gesture recognizing delays between Wi-Fi users using *GestureFlow* and *TCP Relay*.

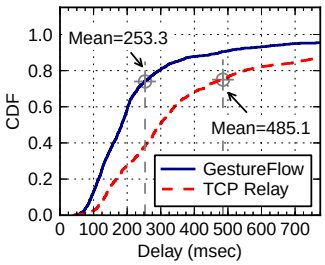


Fig. 14. CDF of gesture recognizing delays between 3G users using *GestureFlow* and *TCP Relay*.

Furthermore, we would like to evaluate the performance of *GestureFlow* in a real-world scenario with four users in "all-to-all" broadcast sessions, with one iPad user connecting to the Internet through campus Wi-Fi, one iPad user using the household Wi-Fi access point, and two iPhone 4S users connecting to the Internet through 3G and EDGE, respectively. Similar to the two-user experiments, these four devices are connected to different ISPs and located in different locations in the same city. Table III summarizes the average gesture recognizing delays in both *GestureFlow* and *TCP Relay* between each pair of devices, over 20 runs of experiments. It is clear that *GestureFlow* achieves better performance: gesture recognizing delays are $23-52\%$ shorter compared to those in *TCP Relay*.

Next, we evaluate an important design choice adopted in *GestureFlow*: the use of multiple paths between the source and each receiver to minimize end-to-end delays. Fig. 15 shows the

TABLE III
COMPARISONS OF GESTURE RECOGNIZING DELAYS (MSEC) USING GESTUREFLOW AND TCP RELAY.

GestureFlow / TCP Relay	Wi-Fi 1	Wi-Fi 2	3G	EDGE
Wi-Fi 1	–	103/178	192/376	309/517
Wi-Fi 2	89/184	–	167/257	274/415
3G	224/391	188/294	–	364/493*
EDGE	347/487	301/428	398/519*	–

Note: there is no direct TCP connection between cellular devices due to NAT restrictions. Relay paths have been used as a result.

average percentage of blocks a node receives from relaying nodes in both *GestureFlow* and *TCP Relay* in the four-user "all-to-all" streaming scenario, along with the 95% confidence interval. We observe that in *GestureFlow*, Wi-Fi and 3G users have more than 10% of the received blocks from relaying nodes, and up to 30% of received blocks are from relaying nodes for the EDGE user, due to longer network delays on direct EDGE links. As a result, EDGE users rely more on relay paths that have shorter delays than those direct ones. However, only a small percentage of blocks are observed from relaying nodes in *TCP Relay*, especially for the EDGE user, which indicates that it fails to take full advantage of multiple paths as *GestureFlow* does.

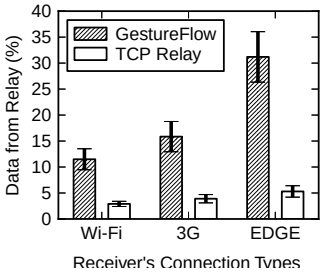


Fig. 15. The percentage of blocks from relaying nodes over all received blocks in *GestureFlow* and *TCP Relay*.

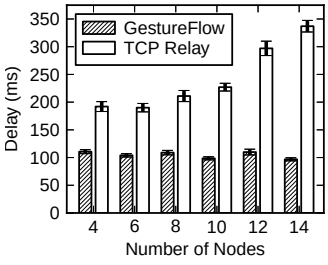


Fig. 16. The average of gesture recognizing delays with different network sizes using *GestureFlow* and *TCP Relay*.

To investigate the scalability of *GestureFlow*, we further study the correlation between the gesture recognizing delay and the number of participating users. Note that all participating nodes are Wi-Fi users in this experiment. Shown in Fig. 16, as the number of nodes increases, the average of gesture recognizing delays in *GestureFlow* varies mildly around 100 ms. We can even observe a slight decrease in the gesture recognizing delay when the number of nodes is large in *GestureFlow*, *e.g.*, 14 nodes, which is due to an increased number of relay paths that may provide shorter end-to-end delays. In contrast, the average of gesture recognizing delays in *TCP Relay* increases significantly when the system scales up, which is mainly due to congested TCP connections that are overwhelmed by relayed blocks. Such an observation implies that a set of complex relay selection and rate control algorithms are required in TCP-based gesture streaming, as opposed to the simpler design of inter-session network coding in *GestureFlow*.

We also investigate the bandwidth overhead in *GestureFlow* by evaluating 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. 17, the gap between these two curves becomes wider as the streaming bit rate becomes 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*.

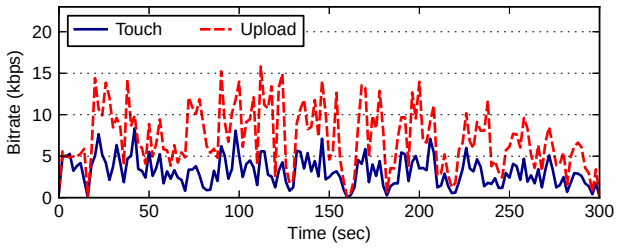


Fig. 17. Bandwidth overhead per user.

B. User Experience Evaluation

Although our experimental results have so far shown that *GestureFlow* is able to provide a better Quality of Experience for gesture streaming in interactive media applications by providing shorter gesture recognizing delays, it may not yet be fully convincing. We are more interested in the actual feedback from real-world users when *GestureFlow* is in use, since it reflects the Quality of Experience directly. As a result, we have conducted a series of experiments to capture user feedback. We invite users to use *MusicScore* in a two-user interactive music composition session over Wi-Fi hotspots. They are asked to rate gesture recognizing delays they have experienced in a 5-min interactive collaboration session, selecting from 4 categories: 1) delay is too long, *i.e.*, not usable from a user's perspective; 2) delay is long, but still tolerable; 3) satisfying, but with a noticeable delay; 4) excellent.

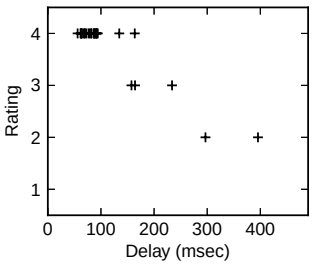


Fig. 18. User experience ratings of different gesture recognizing delays in *MusicScore* using *GestureFlow*.

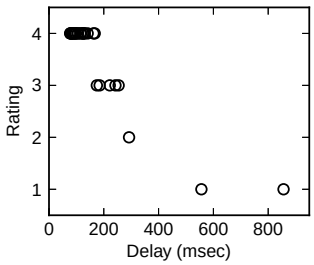


Fig. 19. User experience ratings of different gesture recognizing delays in *MusicScore* using *TCP Relay*.

In Fig. 18 and Fig. 19, we show gesture recognizing delays of collaborative sessions and their corresponding user experience ratings, when *GestureFlow* and *TCP Relay* are in use through Wi-Fi connections, respectively. Clearly, *GestureFlow* is able to garner higher user experience ratings with shorter gesture recognizing delays compared to *TCP Relay*. Similar trends are also observed with other connection types. While most Wi-Fi users reported better collaboration experiences when gesture recognizing delays are shorter than 150 msec, users are observed to be more tolerable to longer delays in 3G networks. Our evaluation results show that users tend to give a rating of 4 even though their experienced delays are around 300 msec in 3G networks. This can be explained as users usually expect 3G networks to be slower than Wi-Fi. Validated by both shorter gesture recognizing delays and higher user experience ratings, we believe that *GestureFlow* is able to achieve a satisfactory Quality of Experience.

C. Performance of Network Coding

Since we apply network coding in *GestureFlow*, it is important to justify this design choice. Fig. 20 shows the relationship between the maximum coding window size W and the gesture recognizing delay. We observe that the gesture recognizing 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.

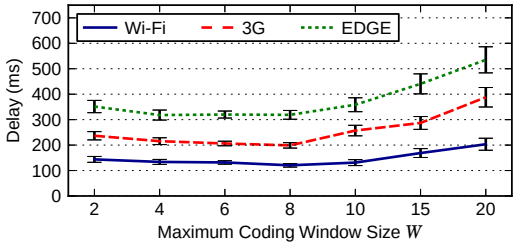


Fig. 20. The average delays along with 95% confidence intervals in different experiment settings.

As an adaptive maximum coding window size adjustment mechanism is specifically designed for QoE-aware gesture streaming, we would also like to verify its effectiveness through performance evaluation. In Fig. 21, we compare the gesture recognizing delays when the adaptive maximum coding window size adjustment mechanism is on and off among Wi-Fi users. Clearly, benefited from the shrunk maximum coding window when gestures are recognized, the average gesture recognizing delay is reduced from 121.5 msec to 102.6 msec. Similar results are also observed in measurements among users with other Internet connection types.

Having evaluated the coding window size and its adaptive adjustment mechanism, we now proceed to observe the actual number of blocks to be coded at each node. We plot the CDF for the number of original blocks and the number of relayed blocks at each node, which are shown in Fig. 22 and Fig. 23, respectively. From Fig. 22, we can see that 90% of time there is no more than 5 original blocks to be coded at a node. This indicates that most of the time there is very little delay added in both the encoding 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 remain idle most of the time, the actual coding window size is naturally reduced. Similarly, Fig. 23 shows that 90% of the coding 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 relaying 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 relaying nodes are useful to downstream receivers with high probability.

Another concern when applying network coding is its CPU load and memory usage, 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, with results shown in Fig. 25. As we can see, the average CPU usage is 8.4%, with peaks corresponding to bursty bit rates in Fig. 17. The dashed line shows the memory usage over time, which is 2.4% on average. iPad nodes have even lower CPU loads as they enjoy a higher CPU frequency in their 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 acceptable.

It is critical to point out that coded blocks in network coding, either from source nodes or relaying nodes, are considered useful only when they are *linearly independent* with one another, or else they are regarded as redundant blocks. The ratio of linear dependence among coded blocks with different coding window size W is also investigated, when evaluating the performance of network coding. For a specific data block, its linear dependence is computed as the percentage of linearly dependent blocks over the total number of coded

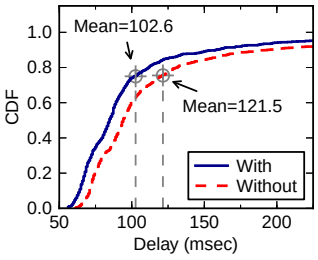


Fig. 21. A comparison of gesture recognizing delays between *GestureFlow* with and without the adaptive coding window size adjustment mechanism.

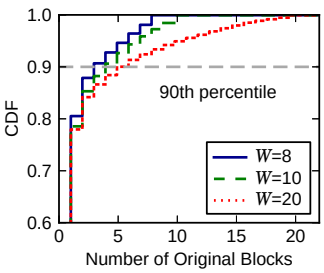


Fig. 22. CDF of the number of original blocks to be coded at a node with different choices of the maximum coding window size (W).

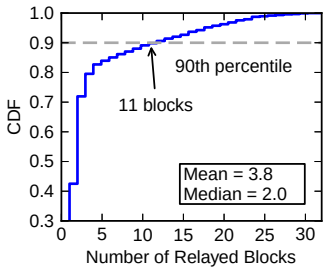


Fig. 23. CDF of the number of relayed blocks to be coded at a node.

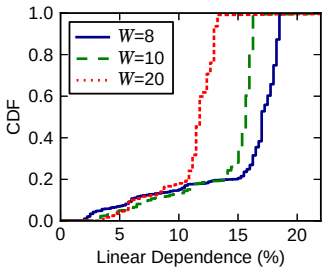


Fig. 24. CDF of linear dependence in different settings of the maximum coding window size.

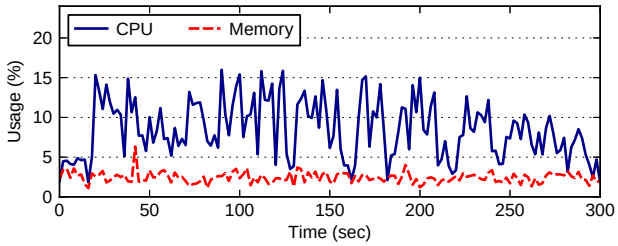


Fig. 25. The CPU load and memory usage of network coding in an iPhone 3GS device.

blocks involving the data block. By plotting the CDF of linear dependence of all coded blocks in our experiment in Fig. 24, we find out that in 90% of them, around 15% of blocks are linearly dependent, which is an alarmingly high percentage. The percentage of linear dependence is even higher when the coding window size is becoming smaller, *e.g.*, it becomes 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 waste due to linearly dependent blocks may escalate with concurrent broadcast sessions. More importantly, a high percentage of linear dependence may result in longer gesture recognizing delays as nodes have to wait for more useful blocks to decode a gesture. With QoE awareness, we need to carefully analyze and address the challenge of linear dependence.

V. The Problem of Linear Dependence

A. Analyzing the Effects of Linear Dependence on QoE

In this section, we show theoretical insights on how linear dependence among coded blocks negatively affects the Quality of Experience of users by increasing their gesture recognizing delays.

We first describe our system model formally. Assume that N users participate in an gesture broadcast session. Each of them is not only a source that generates gestures, but also a receiver and a relay of blocks from other sessions. The network is modelled as a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ is the set of network nodes that represent participating users, *i.e.*, $|\mathcal{V}| = N$. $\mathcal{E}$ is the set of network links. The links are characterized by the total link capacity e_{ij} expressed in blocks per second, and the average packet loss rate π_{ij}, where $(i, j) \in \mathcal{E}$ denotes the link between the nodes i and j in $\mathcal{V}$. We denote the average percentage of linear dependence among coded blocks by ld.

It is obvious that the gesture recognizing delay depends on the average packet loss rate and the percentage of linear dependence among coded blocks. To be exact, the expected delay observed at each node can be computed by estimating the average number of blocks that it receives before it can decode those gestures. Let D_i be the average delay observed at node i for receiving a sufficient number of blocks such that it can decode gestures in coding windows of all other users. D_i has the form of

$$D_i = d_i \sum_{k=(N-1)W}^{\infty} k P_i(k).$$

In this equation, k is the number of blocks that node i receives before it can decode the gestures; $P_i(k)$ denotes the probability of decoding these gestures after receiving exactly k blocks; the constant d_i denotes the average delay for receiving one block and can be approximated as $d_i = \frac{1}{\sum_{j \in \mathcal{V}_{-i}} e_{-i}}$, where $\mathcal{V}_{-i}$ is the set of nodes in $\mathcal{V}$ without node i, *i.e.*, $\mathcal{V}_{-i} = \mathcal{V} \setminus \{i\}$. Note that k includes all coded blocks that can be either linearly dependent or independent from other blocks.

Since at most W original blocks, including all received coded blocks from other $N-1$ broadcast sessions, are allowed to be coded together to produce a new coded block at each node, the minimum number of blocks needed for decoding gestures from all other users equals to $(N-1)W$. That is to say, the probability of decoding with fewer blocks than $(N-1)W$ equals 0. Hence, the probability $P_i(k)$ of decoding gestures from all other users with exactly k blocks corresponds to the probability of forming a full rank system upon receiving the k^{th} block but not before that. Analytically,

$$P_i(k) = \binom{k-1}{k-(N-1)W} p_i^{(N-1)W} (1-p_i)^{k-(N-1)W},$$

where p_i represents the probability that a useful block arrives at node i.

Since a block is considered useful if it is not lost due to packet erasures and it is linearly independent to coded blocks that a node has already received, the probability p_i can be represented by the link capacity, the packet loss rate on each link, as well as the average percentage of linear dependence among coded blocks. It takes the following form:

$$p_i = \frac{(1-ld)\sum_{j \in \mathcal{V}_{-i}} e_{ji}(1-\pi_{ji})}{\sum_{j \in \mathcal{V}_{-i}} e_{ji}}.$$

More formally, the probability that a useful block arrives at each node is defined as the fraction of total useful blocks arrived over its incoming bandwidth capacity.

From our analysis, we can see that when the percentage of linear dependence among coded blocks ld increases, the probability p_i of a useful block arriving at each node will decrease. This then results in an increase of probability $P_i(k)$, since $\frac{\partial P_i(k)}{\partial p_i} < 0$. As a consequence, the average gesture recognizing delay D_i observed at each node will increase.

B. Mitigating Linear Dependence

To mitigate the high percentage of linearly dependent blocks that incur longer gesture recognizing delays, we are inspired by systematic Reed-Solomon codes, and propose to generate coding coefficients for the original blocks at each node based on the Vandermonde matrix in *GestureFlow*.

With coding coefficients generated by the Vandermonde matrix, each node codes original blocks first, rather than codes coded blocks belonging to its own session 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 coding all original blocks, a node starts to generate and code coded blocks from its own session. In order to code k original blocks using an (n, k) Vandermonde matrix over a Galois field F_q, a node is able to generate up to $n - k$ coded blocks, after original blocks are coded. In *GestureFlow*, a $(n-k)\times k$ Vandermonde matrix G [3] 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 blocks coded from the source is guaranteed with the use of the Vandermonde matrix. These blocks from the source are always innovative once received.

Compared with random linear codes, one difference of using the Vandermonde matrix at the source is that it is not *rateless*. With the $(n - k) \times k$ Vandermonde matrix G, a maximum of $n - k$ coded blocks can be coded, 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. IV-C 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, relay paths, and inter-session network coding are still adopted. Note that each node still uses random linear network coding to generate coding coefficients when recoding received blocks, so that there are no restrictions imposed on the actual number of coded blocks in the coding process.

C. Evaluating the Use of the Vandermonde Matrix

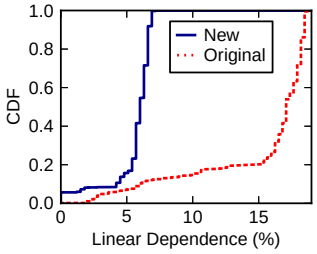


Fig. 26. CDF of linear dependence with the Vandermonde matrix used at the source nodes ($W = 8$).

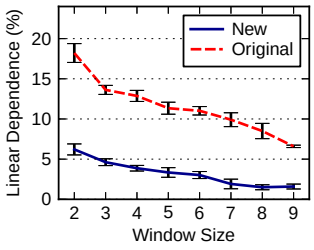


Fig. 27. Linear dependence with different coding window sizes.

The effectiveness of using the Vandermonde matrix at the source in *GestureFlow* is evaluated with additional experiments with *MusicScore*. Fig. 26 shows the comparison of CDFs of linear dependence between the original *GestureFlow* design and the use of Vandermonde matrix to mitigate linear dependence. The maximum coding window size W is set to be 8. It is clear that the 90^{th} percentile of linear dependence is significantly reduced by using the new design, from 18% to 6%. Since blocks 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 relaying nodes, which is acceptably low. The ratio of linear dependence with different coding window sizes is explored in Fig. 27. We can see that the ratio of linear dependence is decreasing as the coding window size increases, and the ratio with the Vandermonde matrix 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 gesture recognizing 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 IV. We can observe that by using Vandermonde matrix as coding coefficients at the source, average gesture recognizing delays through different kinds of connections have all been evidently reduced. Since the redundancy due to linear dependence is mitigated with the Vandermonde matrix at the source, a received block can be used to decode with a higher probability, which reduces the decoding delay.

TABLE IV
GESTURE RECOGNIZING DELAYS (MSEC) AT WI-FI 1 USER WITH NEW GESTUREFLOW DESIGN.

$(\overline{x}, s)$	Wi-Fi 2	3G	EDGE
New	$(89, 23)$	$(177, 87)$	$(287, 168)$
Original	$(103, 48)$	$(192, 104)$	$(309, 191)$

In summary, our experiments in Sec. IV and Sec. V have evaluated our important design decisions made in *GestureFlow*, with the objective of reducing gesture recognizing delays. Our results have confirmed that gesture recognizing delays are effectively reduced with our proposed protocol in *GestureFlow*, and that it scales well when the number of participating nodes increases.

VI. Related Work

With the inception of network coding [4] and random network coding [1], [2] in information theory, the topic has attracted a substantial amount of research attention. Analytical studies [2], [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 experimental 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. UUSee, Inc. has successfully adopted network coding into its commercial peer-assisted on-demand streaming protocol [9]. Different from these applications, 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, RTP [10] and RTSP [11] are able to provide one-to-all delivery of data with real-time properties over IP multicast. Since RTP and RTSP are originally designed for IP unicast, reliable multicast protocols [12], [13] were proposed to improve the performance of multi-party streaming, by reducing ACK/NAK implosion in back traffic and optimizing retransmissions on multicast channels. In addition, network coding has been applied to incorporate with transport protocols. For example, CodeCast, presented by Park *et al.* [14], is a network coding based multicast protocol for low-latency multimedia streaming. Sundararajan *et al.* [15] 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 interactive broadcast sessions, each involving a stream of gesture events, over regular IP unicast. 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.

The performance improvement brought by inter-session network coding has drawn some recent research attention in the literature. Eryilmaz *et al.* provide a theoretical framework in which a dynamic routing-scheduling-coding strategy is proposed to decide whether blocks from two sessions should be coded together at a node [16]. Yang *et al.* propose to divide multiple sessions into groups and construct a linear network coding for each group, with consideration of improving the system's benefits on bandwidth and throughput [17]. Focused on directed networks with two multicast sessions, Wang *et al.* discuss various aspects of pairwise inter-session network coding, including the sufficiency of linear codes and the complexity advantages of identifying coding opportunities [18]. I^2NC combines inter-session and intra-session network coding to improve the throughput in lossy wireless environments [19]. In contrast to previous research, the primary objective in *GestureFlow* is to reduce the gesture recognizing delay as a QoE metric in interactive multimedia applications, which has not been the focus of study in previous work.

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

[2] 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 Int'l Symposium on Information Theory (ISIT)*, 2003, p. 442.

[3] R. M. Roth, *Introduction to Coding Theory*. Cambridge University Press, 2006.

[4] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung, "Network Information Flow," *IEEE Trans. Info. 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*, 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. Int'l WWW Conference*, 2007, pp. 903–912.

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

[9] Z. Liu, C. Wu, B. Li, and S. Zhao, "UUSee: Large-Scale Operational On-Demand Streaming with Random Network Coding," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[10] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889, 1996.

[11] H. Schulzrinne, A. Rao, and R.Lanphier, *Real Time Streaming Protocol (RTSP)*. RFC 2326, 1998.

[12] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing," *IEEE/ACM Trans. Networking*, vol. 5, pp. 784–803, 1997.

[13] S. Paul, K. Sabnani, J.-H. Lin, and S. Bhattacharyya, "Reliable Multicast Transport Protocol (RMTP)," *IEEE J. Sel. Areas Comm.*, vol. 15, no. 3, pp. 407–421, 1997.

[14] 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.

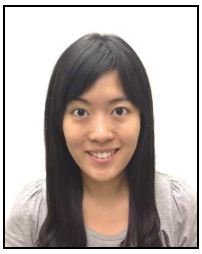
[15] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, "Network Coding Meets TCP," in *Proc. IEEE INFOCOM*, 2009, pp. 280–288.

[16] A. Eryilmaz and D. S. Lun, "Control for Inter-Session Network Coding," in *Proc. Workshop on Network Coding (NetCod)*, 2007.

[17] M. Yang and Y. Yang, "A Linear Inter-Session Network Coding Scheme for Multicast," in *Proc. 7th IEEE Int'l Symposium on Network Computing and Applications*, 2008, pp. 177–184.

[18] C.-C. Wang and N. Shroff, "Pairwise Inter-Session Network Coding on Directed Networks," *IEEE Trans. Info. Theory*, vol. 56, no. 8, pp. 3879–3900, 2010.

[19] H. Seferoglu, A. Markopoulou, and K. Ramakrishnan, "I2NC: Intra- and Inter-Session Network Coding for Unicast Flows in Wireless Networks," in *Proc. IEEE INFOCOM*, 2011, pp. 1035–1043.



Yuan Feng received her B.Engr. from the School of Telecommunications, Xidian University, Xi'an, China, in 2008, and her M.A.Sc. degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2010. She is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Toronto. Her research interests include optimization and design of large-scale distributed systems and cloud services. She is a student member of IEEE.

Zimu Liu received his B.Engr. degree from the Department of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China, in 2007, and his M.A.Sc. degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2009. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Toronto. His research interests include design and implementation of reusable mobile frameworks, measurement studies on large-scale peer-to-peer streaming systems, and applications of network coding. He is a student member of IEEE.

Baochun Li received the B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Nortel Networks Junior Chair in Network Architecture and Services from October 2003 to June 2005, and the Bell University Laboratories Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale multimedia systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. Dr. Li was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a member of ACM and a senior member of IEEE.