Forward Error Correction for Fast Streaming with Open-Source Components

John Reynolds
Faculty Advisor: James S. Plank
Collaborator: Diederick Huijbers (software development, Netherlands)
The University of Tennessee at Knoxville

Unreliable networks, implemented with wired or WiFi connections, cause a large amount of packet loss in a multitude of different networked applications. An example of such an application is video streaming. When video data packets are transmitted over lossy channels, the receiving user may not receive all of the packets that were sent. This data loss results in a bad streaming experience if it is not handled properly. The NACK (negative acknowledgement) protocol is used by many applications to request the retransmission of lost data; however, with FEC (Forward Error Correction) [5] the need for the NACK protocol and retransmission is greatly lessened (or rendered entirely unnecessary). In this project, we have built a video streaming environment to explore the effectiveness of FEC, based on erasure codes and implemented in software, across unreliable networks. We have focused on open-source components for video conversion, real-time networking, FEC coding, and adherence to standardized protocols for inter-component communication. In this paper, we briefly outline the software structure of our project, and demonstrate the effectiveness of FEC for video streaming.

Forward Error Correction is an efficient solution to the packet loss problem. Data source symbols being sent across a network may be encoded using erasure codes to create redundant data. This redundant data may then be decoded on the receiving end to recover dropped packets. Many different erasure codes exist. The FEC component of this video streaming library has been implemented using Reed-Solomon based codes from an open-source erasure coding library, Jerasure [1]. Reed-Solomon codes fall under the category of erasure codes labeled as MDS (Maximum Distance Separable). MDS codes have a nice property: k source symbols can be encoded to create m repair symbols such that among the k+m total symbols, any m erasures are tolerable. This means that data recovery is possible for any combination of m packets lost as long as at least k packets are received. The type of packet(s) received (source or repair) does not matter.

The flow of data through the video streaming library we created is handled by many components. Video frames are captured using the “video_capture” library developed by Diederick Huijbers to discover hardware camera devices, activate them, and collect their video frame data output. These video frames are then converted to the appropriate YUV format by Google’s open-source YUV conversion & formatting library, libyuv. Once converted, the frames are encoded to VP8 video format using Google’s libvpx codec. The VP8 encoded video frame is broken into packet sized chunks and transmitted as payload data [4] in UDP/RTP packets [3]. If any of the k packets are dropped in transmission, the related VP8 frame cannot be decoded. The component dealing with packet loss before FEC was added to this project performed NACK-style retransmission. With FEC encoding enabled, the UDP/RTP/VP8 packets become the k source symbols. Jerasure is used to perform Reed-Solomon encoding. The encoding of source symbols produces m repair symbols and adheres to RFC 6865 [2]. The k source symbols and m repair symbols are sent to the receiver. If the receiver detects packet loss, the source and repair symbols received (assuming k symbols arrived) are decoded to retrieve the lost VP8 frame data. If no packets are lost and all k source symbols arrive, the Jerasure decoding step is skipped. Once all of the VP8 frame data is available, whether from received k source symbols or from Jerasure decoding, the data is merged back together from the packet sized chunks and decoded using libvpx to restore the original video format. The original YUV frame data is then bound to an OpenGL texture and rendered.
Real-time streaming data is typically sent in UDP (User Datagram Protocol) packets using the RTP (Real-Time Transport Protocol) packet format [3]. Because of the nature of UDP, packets are not guaranteed to arrive (unlike TCP). FEC renders NACK-style retransmission of lost VP8 frame data unnecessary in this project. Rather than having to request lost video data from the sender, which imposes the time constraints of re-transmission upon real-time streaming, we simply decode using Jerasure to reconstruct lost packets and rebuild the frame immediately without communication with the sender. FEC has allowed us to remove the NACK protocol for handling packet loss.

In real-time video streaming, the number of frames sent per second must match closely the number of frames received and rendered per second. We performance tested by sending and receiving a video feed on two separate wireless networks with 20% packet loss. The number of frames sent per second and the number of frames received and rendered per second both averaged around 20 fps. While there is plenty of room for optimization in the application, this figure is significant. This means that the receiving application is able to receive and decode frames at an average rate that is the equal to the average rate of frames being sent. The Jerasure decoding time to recover lost packets is negligible: 0.00008 seconds on average with an average frame size of 2573 bytes and an average of 4.6 RTP/VP8 packets per frame. The impact of applying FEC to video streaming is negligible concerning added computational complexity.

The development of this open-source video streaming library has provided a testbed. This testbed will allow us to explore the usage of other erasure codes, for example, Raptor codes [6], which have been designed to react to changing network conditions. We may also modify the k and m Reed-Solomon parameters based on statistics sent from the receiver to the sender periodically. Feedback can also be used to configure the coding parameters based on network conditions (reactively). We hope that our experience with video streaming will allow us to design a general purpose open source library for software-based FEC.

References:


