Broadcasting Video Content in Dense 802.11g Sports and Entertainment Venues

Project Update

Last update: 8/29/2010

Principal Investigators

Dr James Martin
School of Computing
Clemson University
Clemson, SC
jim.martin@cs.clemson.edu
Phone: 864 656 4592

Dr James Westall
School of Computing
Clemson University
Clemson, SC
westall@cs.clemson.edu
Phone: 864 656 5868

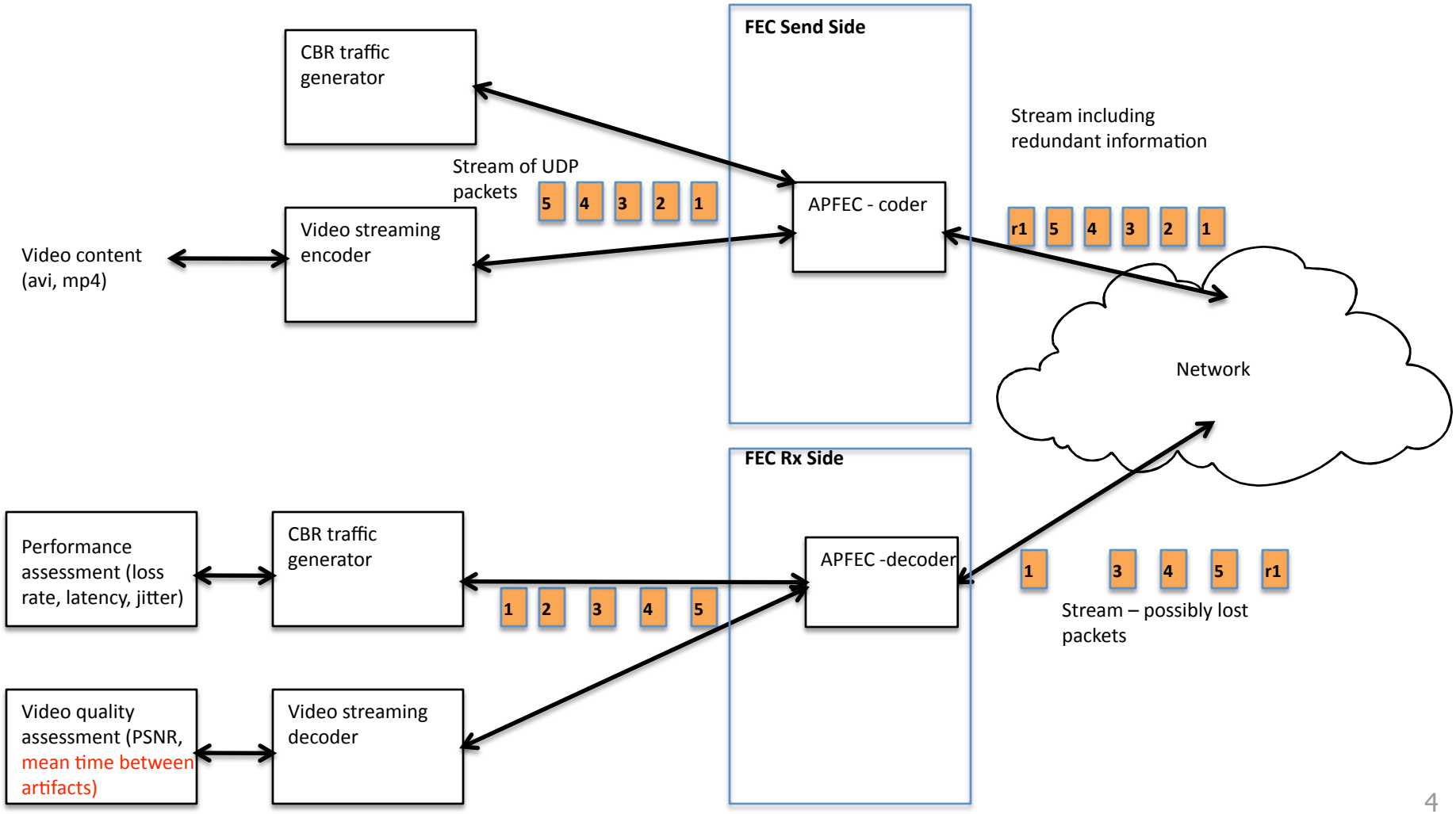
Project Progress: Developing the Models

- 802.11 Protocol
  - ✓ 802.11g PHY behaviors: refer to Experiments 1 and 2 documented in projectUpdate-7-8-2010.pdf
  - ✓ Adaptive modulation/coding: model supports basic rate and one statically defined rate for data – it's possible to add AMC in the future
  - ✓ Multicast: model supports multiple multicast streams – broadcasts at basic rate over the air (under test)
- Packet loss process
  - Loss metrics: mean burst loss length (MBL), mean interloss distance (MILD)
  - Model-based loss: Bernoulli loss, Hidden Markov models, Gilbert-Elliot model
  - Propagation effects: propagation models, Rx packet capture
  - Protocol effects: congestion, collisions
- Application:
  - Video streaming traffic generator (encoder/decoder)
  - ✓ Application FEC Model
  - Application oriented quality assessment

Outline of this Status Update

- Overview the design of the application FEC traffic generator
- Demonstrate results using a simple unicast CBR-based stream

Application FEC Model



APFEC Video Streaming

- Send side highlights
  - Configured with an N and K
  - Maintains an index identifying the block number
    - Maintained as an unsigned integer, first block of a new session is always 1 (counter wraps at 2EXP32). This is sent in each packet.
  - Redundant packets always sent at the end of the block
  - Variable size data packets:
    - Find the average packet size of packets sent in the block. Set all FEC packets to this size
  - BlockTimer pops if K data packets don't arrive within a timeout amount of time. Two choices:
    - Pad the block to ALWAYS send N packets
    - Scale down the N and K for this block (this is what the code currently does)
  - Each packet will contain:
    - blockNumber, blockIndex (place in the current block), and the N and K that are in effect for the current block
    - Only data packets have sequence numbers, FEC ordering is done using the blockNumber and blockIndex.

APFEC Video Streaming

- Rx side highlights
  - N and K for a given block are determined on a block by block basis (from the N_ and K_ contained in each packet)
  - blockTimer required in the event K packets do not arrive in a timely manner
  - Two queues are maintained
    - blockQ: queues all packets that arrive that are associated with the current block being collected
    - OutOfOrderQ : queues all packets that arrive that are NOT associated with the current block being collected
  - In order packets that arrive are added to the blockQ and a duplicate packet is created and passed up to the application
  - When 1 or more data packets in the current block do not arrive, the block will be completed once any of the following occurs:
    - Once K or more packets have arrived
    - Once a threshold number of packets from next blocks have arrived
    - Once the blockTimer pops

Model Object Design

```
class AppFecAgent : public Agent {
public:
   AppFecAgent();
   virtual void sendmsg(int nbytes, const char *flags = 0)
   {
      sendmsg(nbytes, NULL, flags);
   }
   virtual void init(int streamID, int N, int K);
   virtual void sendmsg(int nbytes, AppData* data, const char *flags = 0);
   virtual void recv(Packet* pkt, Handler*);
   virtual void blockTimerHandler(int);
   virtual int command(int argc, const char*const* argv);
   virtual void dumpFinalStats(char *outputFile);
```

BlockTimerObject
Start(delay)
Stop()
Handler()

```
int blockindex_;
   int numberBytesInBlock;
   int numberPacketsInBlock;
   int streamID;
   int seqno_;
   int blockno_;
  int N_;
   int K_;
   int typeFEC;
   int streamType;
   //For creating the output trace files
   char traceString[32];
   char *tptr;
   FILE* fp;
   int state;   //-1 uninited, 1 inited
   double blockTimerDelay;
   int numberFECPackets;

   AppFecBlockTimer *myTimer;
```

```
class AppFecSourceAgent : public AppFecAgent {
public:
   AppFecSourceAgent();
   virtual void init(int streamID, int N, int K);
   virtual void sendmsg(int nbytes, const char *flags = 0)
   {
      sendmsg(nbytes, NULL, flags);
   }
   virtual void sendmsg(int nbytes, AppData* data, const char *flags = 0);
   virtual void blockTimerHandler(int);
   virtual int command(int argc, const char*const* argv);
   virtual void sendFEC(int numberFECPackets);
   virtual void dumpFinalStats(char *outputFile);
protected:
};
```

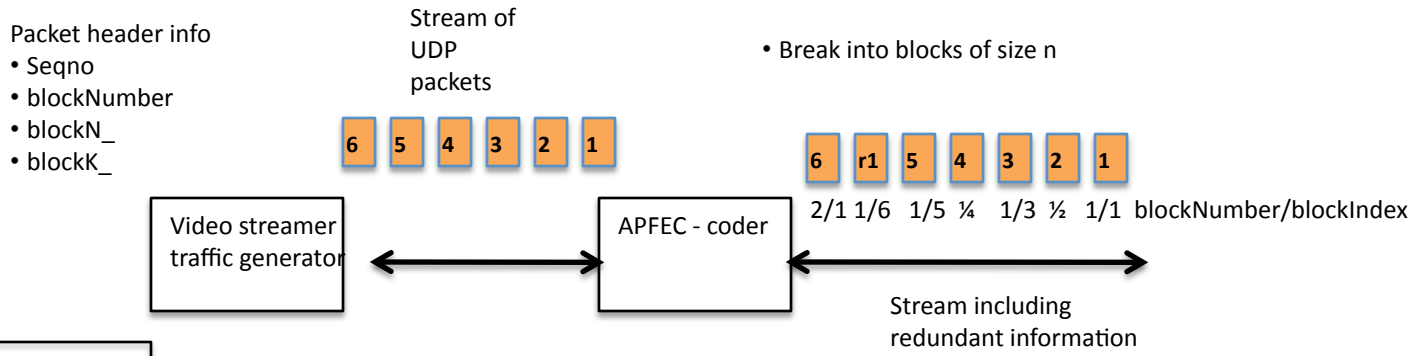
```
class AppFecSink : public AppFecAgent {

public:
  AppFecSink();
  virtual void init(int streamID, int N, int K);
  void recv(Packet* pkt, Handler*);
  void passPacketUp(Packet* pkt);
  virtual void blockTimerHandler(int);
  virtual int command(int argc, const char*const* argv);
  virtual void dumpFinalStats(char *outputFile);
  void processFECBlock();
  int transferBlock(OrderedListObj *srcList, OrderedListObj *dstList, int
currentBlockNumber);
```

```
OrderedListObj *blockQ;
   OrderedListObj *OutOfOrderQ;
   int clearQueue(OrderedListObj *ListPtr);
   int blockState;
   int maxBlockListSize;
   int maxOutOfOrderListSize;
   int nextExpectedSeqNumber;
   int currentBlockNumber;
   unsigned int numberInOrderPacketArrivals;
   unsigned int numberOutOfOrderPacketArrivals;
   unsigned int numberBlockQDrops;
   unsigned int numberOutOfOrderQDrops;

   int OutOfOrderQThreshold;
```

Application FEC Send Side

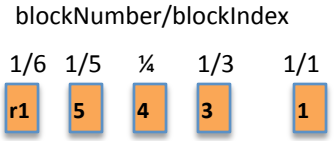



```
constructor:
 bind("N_", &N_);   //tcl param
 bind("K_", &K_);   //tcl param
 blockTimerDelay = 0.200;
```

```
* Function: void AppFecSourceAgent::sendmsg(int nbytes, AppData* data, const char* flags)
*
*  pseudo code:
*  set n to number of packets required to send the message
*  for n packets
*  {
*    if this packet is the FIRST data packet in a block
*      start the blockTimer
*
*    increment the seqno_
*    increment the blockIndex_
*    allocate and setup the packet (set contentType to VIDEO)
*    send the packet
*
*    if (blockIndex_ >= K_)  (if this packet is the LAST data packet in a block)
*      call sendFEC(numberFECPackets) to send the FEC data
•      stop the blockTimer
•    }
```

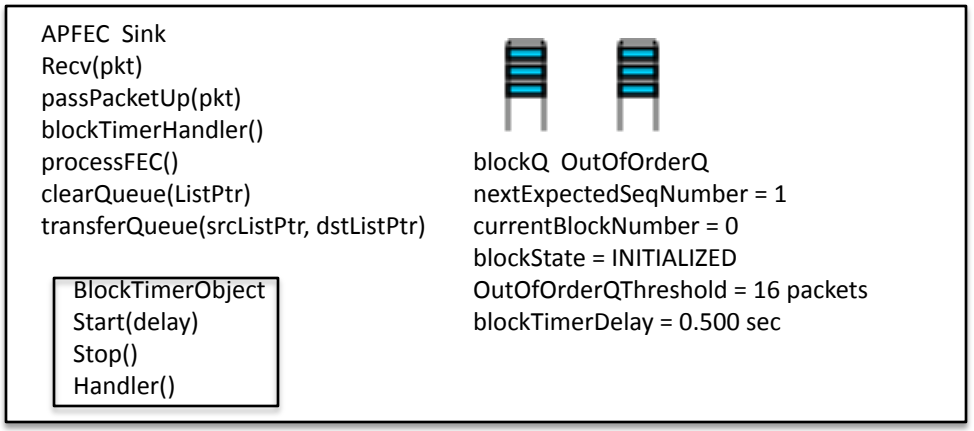
```
void AppFecSourceAgent::sendFEC(int numberFECPackets)
*This method completes the FEC process for the current block
*  If do not have a full block
*    if fixedBlockMode_ is FALSE
*      scale N_ and K_ (numberFECPackets might be reduced)
*    else
*      keep K_ and N_ to the configured values
*  n = updatedN_- updatedK_;
*  find avg size of data pkt sent in block (nBytesPerPacket =
                    numberBytesInBlock /  numberPacketsInBlock)
*  For n packets
*  {
•     create a packet (size : nBytesPerPacket)
•    increment the blockindex_ (do not increment seqno_)
•     setup the packet (rh->contentType = FEC_OVERHEAD)
*     send the packet
*  }
```

Application FEC Rx Side



Packet header info
- Seqno
- blockNumber
- blockIndex
- blockN_
- blockK_

- Recv(pkt) :
- * pseudo code (given packet pkt):
- * if configured for no fec
- * pass pkt up
- * else if pkt associated with prior blocks
- * throw out packet as it's a dup or not needed
- * else if pkt is in the current block
- * if pkt is a duplicate, just throw it away
- * return;
- * if start of new block
- * start block timer
- * state = in progress
- * insert in blockQ
- * if pkt is data, clone packet and passPacketUp(pkt)
- * if current block is complete (if at least K_ arrived)
- * process FEC block
- * else if pkt is NOT associated with current block
- * if pkt is a duplicate, just throw it away
- * return
- * insert in OutOfOrderQ
- * if size of OutOfOrderQ exceeds threshold
- * process FEC block //handles transferring Queues
- *



blockTimerHander()
*numberBlockTimeouts++;
*processFECBlock();

clearQueue(ListPtr)

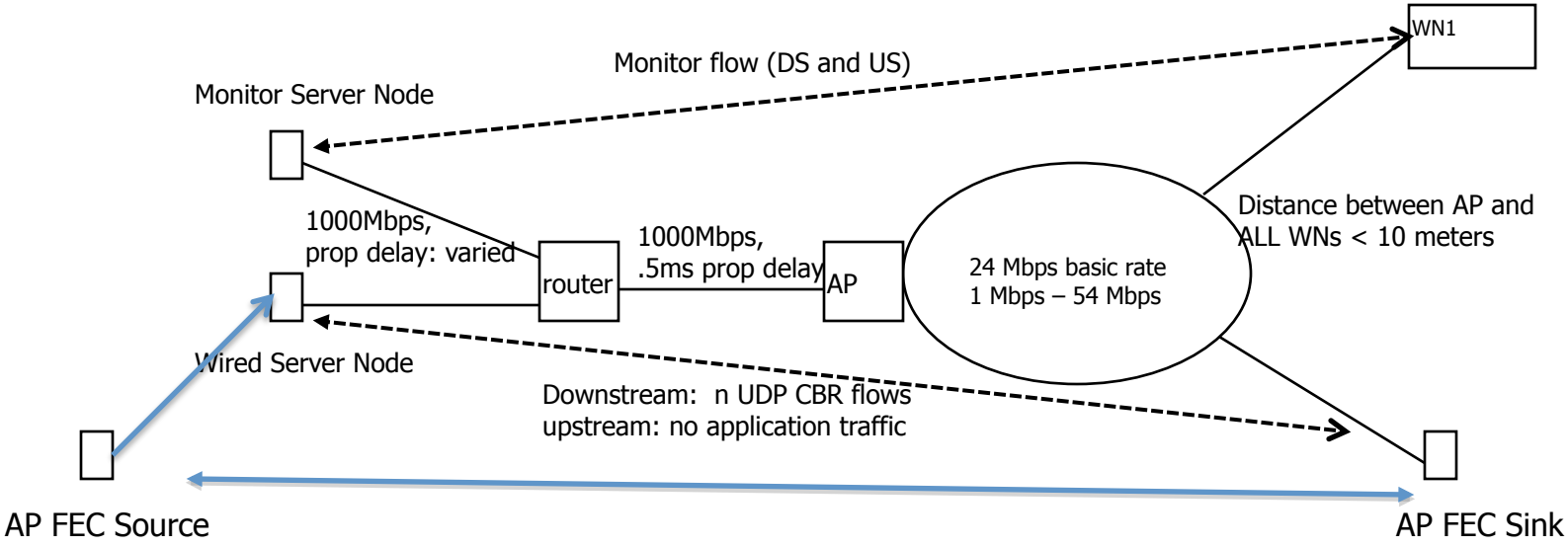
passPacketUp(pkt)

transferQueue(srcListPtr,dstListPtr)

Application FEC Rx Side

```
*processFEC()
* while (blockQ->getListSize() > 0)
*  {
*    stop block timer
*    set blockState to completed
*    get this blocks N and K
*    if N_ is 1, this is an error
*    if  all N_ received
*      loop through the blockQ
*      {
*        if an element is not yet sent up, clone it and passPacketUp(pkt)
*        delete the element
*      }
*    if K_ or more were received
*      loop through the blockQ
*      {
*        if the element is in order
*          if it has not been sent up yet, clone it and passPacketUp(pkt)
*          delete the element from the Q
*        if an element is missing in the Q, regenerate and passPacketUp(pkt)
*      }
*    if less than K_ were received
*      loop through the blockQ
*      {
*         if an element is not yet sent up, clone it and passPacketUp(pkt)
*         delete the element
*      }
*    advance currentBlockNumber++;
*    reset block state variables such as numberPacketsInBlock and numberBytesInBlock
*    if blockQ not empty, throw HARD ERROR
*    if OutOfOrderQ not empty
*      transfer any of the next block (if it exists) from the Outoforder to the blockQ
*      if transfered > 0
*        start block timer
*        set blockState to in progress
*        Get current blocks K
*        if (blockQ->getListSize < K_)
*         break;
*  }
```

Experiment 3 (EXP3)



Focus on Validating the Application FEC traffic generator. There is a single unicast CBR application flow (CBR rate 3Mbps). The main experimental parameters:

- N,K, and q (redundancy factor: N-k / N)
- Loss process: for now limited to a simple bernoulli process
- Packet size: this is the packet size before overhead. Each packet overhead will include octets for the following headers
  - IP : 20 octets
  - UDP: 8 octets
  - RTP : 32 octets

Results

Experiment Definition

Experiment	Description	Efficiency (FEC)	Efficiency (Total)	Effective Loss Rate	Average Packet Latency
3.1	N,k: 10,8, vary loss rate (p)	Experiment Results			
3.2	Vary N, hold k at 5, p=0.20				
3.3	Vary (N,k) – increase blocksize, maintain q of 0.20				
3.4	N,k:10,8, vary packet size				

Experiment Results:

- Efficiency (FEC): assesses efficiency of just FEC: totalDataBytesPassedUp / totalDataBytesReceivedBySink
  - Note: both stats include all packet overhead and therefore cancels out in the ratio
- Efficiency (Total) : assess overall efficiency: Observed Application Goodput / Total raw BW consumed by flow
  - Note:goodput measure is application data and does not include packet overhead.
- Effective loss rate: Receiver's estimate of loss count / Receiver's estimate of total amount of data (including FEC) sent
- Average packet latency: end-to-end latency averaged for all packets in the flow

Results

Experiment 3.1: vary p

(N,K, q)	Loss Rate (bernoulli)	IP Packet Size (bytes)	Efficiency (FEC)	Efficiency (Total)	Effective Loss Rate	Average Packet Latency (seconds)
10,8, 0.2	0.05	1520	0.835	0.769	0.009	0.006743
10,8, 0.2	0.10	1520	0.863	0.768	0.034	0.012766
10,8, 0.2	0.15	1520	0.879	0.768	0.074	0.022222
10,8,0.2	0.20	1520	0.885	0.769	0.125	0.035516
10,8,.02	0.25	1520	0.878	0.769	0.181	0.049428
10,8, 0.2	0.30	1520	0.865	0.768	0.249	0.064520

Results:

- Efficiency (FEC): assesses efficiency of just FEC: totalDataBytesPassedUp / totalDataBytesReceivedBySink
  - Note: both stats include all packet overhead and therefore cancels out in the ratio.
- Efficiency (Total) : assess overall efficiency: Observed Application Goodput / Total raw BW consumed by flow
  - Note:goodput measure is based on application data and does not include packet overhead.
- Effective loss rate: Receiver's estimate of loss count / Receiver's estimate of total amount of data (including FEC) sent
- Average packet latency: average end-to-end latency averaged for all packets in the flow.

Results

Experiment 3.2: k=5, vary N

(N,K, q)	Loss Rate (bernoulli)	IP Packet Size (bytes)	Efficiency (FEC)	Efficiency (Total)	Effective Loss Rate	Average Packet Latency (seconds)
5,5, 0.000	0.20	1520	1.000	0.959	0.200	0.004685
6,5, 0.167	0.20	1520	0.888	0.800	0.14	0.041270
7,5, 0.286	0.20	1520	0.822	0.687	0.094	0.019975
8,5, 0.375	0.20	1520	0.749	0.599	0.067	0.011193
9,5, 0.440	0.20	1520	0.682	0.532	0.051	0.007785

Results

Experiment 3.3: increase N,k, fixed q=0.2, p=0.2

(N,K, q)	Loss Rate (bernoulli)	IP Packet Size (bytes)	Efficiency (FEC)	Efficiency (Total)	Effective Loss Rate	Average Packet Latency (seconds)
10,8, 0.2	0.20	1520	0.885	0.769	0.125	0.035515
15,12, 0.2	0.20	1520	0.890	0.768	0.122	0.041608
20,16, 0.2	0.20	1520	0.899	0.769	0.118	0.042752
25,20,.02	0.20	1520	0.900	0.768	0.114	0.051264
30,24, 0.2	0.20	1520	0.904	0.768	0.113	0.061059

Results

Experiment 3.4: vary packet size

(N,K, q)	Loss Rate (bernoulli)	IP Packet Size (bytes)	Efficiency (FEC)	Efficiency (Total)	Effective Loss Rate	Average Packet Latency (seconds)
10,8, 0.2	0.20	316	0.854	0.625	0.126	0.009881
10,8, 0.2	0.20	572	0.866	0.701	0.125	0.015243
10,8, 0.2	0.20	828	0.871	0.731	0.124	0.020493
10,8,.02	0.20	1084	0.876	0.747	0.123	0.025807
10,8, 0.2	0.20	1260	0.876	0.754	0.124	0.029866