Lab 2 - Codes for Efficient Transmission of Data

Authors:

v1.0 (2014 Fall) Kangwook Lee, Kannan Ramchandran

v1.1 (2015 Fall) Kabir Chandrasekher, Max Kanwal, Kangwook Lee, Kannan Ramchandran

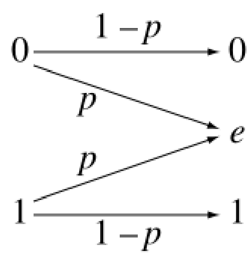
v1.2 (2016 Spring) Kabir Chandrasekher, Tony Duan, David Marn, Ashvin Nair, Kangwook Lee, Kannan Ramchandran

v1.3 (2018 Spring) Tavor Baharav, William Gan, Alvin Kao, Kannan Ramchandran

Introduction ¶

When sending packets of data over a communication channel such as the internet or a radio channel, packets often get erased. Because of this, packets must be sent under some erasure code such that the data can still be recovered. In CS 70, you may have learned about an erasure code that involves embedding the data in a polynomial, and then sampling points from that polynomial. There, we assumed that there were at most k erasures in the channel. This week, we'll explore a different channel model in which each packet independently has a probability p of being erased. In particular, this lab will look at random bipartite graphs (the balls and bins model).

A little more on the channel and the erasure code; formally, our channel is called the binary erasure channel (BEC), where bits that are sent through a noisy channel either make it through unmodified or are tagged as "corrupt", in which case the recieved information is dropped in all further information processing steps. Here's an image that shows what happens.



If we wanted to convey a message, we could consider a feedback channel in which the receiver tells the sender which messages were received and the sender re-sends the dropped packets. This process can be repeated until the receiver gets all of the intended message. While this procedure is indeed optimal in all senses of the word, feedback is simply not possible in many circumstances. If Netflix is trying to stream a show chunked into n data chunks to a million people, its servers can't process all the feedback from the users. Thus, Netflix must use a method independent of feedback. If they use near-optimal codes to encode and constantly send out the same random chunks of the video's data to all users, then they can be sure that users get what they need in only a little more than n transmissions

no matter what parts of the show each individual user lost through their specific channel!

So what's the secret to this magic? It's a two step process of clever encoding and decoding:

Encoding

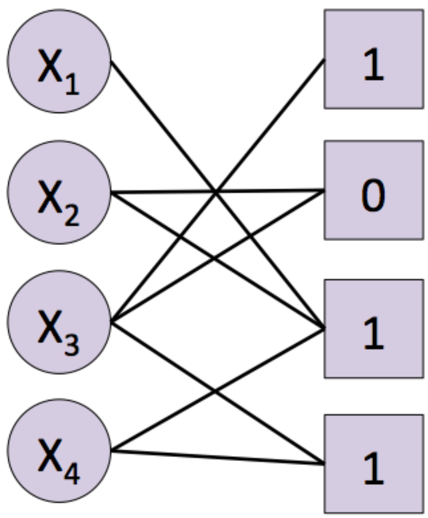
1. Suppose your data can be divided into n chunks. First, pick an integer d $(1 \leq d \leq n)$ according to some distribution.
2. With d picked, now select d random chunks of the data and combine their binary representations together using the XOR operator.
3. Transmit these chunks, along with the metadata telling which actual chunk indices were XOR'd, as a packet. If a packet is erased, both the chunks it contains and the chunk indices would be lost.

Decoding

1. For each packet that has been received, check if it only contains one chunk, in which case the packet is exactly equal to the single chunk it contains. If not, we can check if any of the chunks in the packet are already known, in which case XOR that chunk with the packet and remove it from the list of chunk indices that make up the packet.
2. If there are two or more indices in the list left for the packet we cannot figure out any more information! Put it on the side for looking at later.
3. With any newly decoded information, we may be able to decode previously undecodable packets that we had put on the side. Go through all unsolved packets and try to decode more packets until nothing more can be done.
4. Wait for the next packet to come and repeat!

Now what's left for you to do? Well, remember that number d? It needs to be picked according to some distribution, and which distribution is the million dollar question!

Example



Consider the above bipartite graph. Here, the right square nodes represent the packets, and the left circular nodes represent the data chunks $(X_i, i = 1, .., 4)$. There is an edge from a packet to a chunk if the packet contains that chunk. Let's try decoding the packets chronologically.

1. Since the first packet contains only the third data chunk, we are able to immediately resolve it and find that $X_3 = 1$.
2. The second packet contains the second and third chunks XOR'd together. Since we already know the third chunk however, we can XOR the third chunk ($X_3 = 1$) with the data packet (0) to get the value of the second data chunk, $X_2 = 1$.
3. The third packet contains the XOR of data chunks 1, 2, and 4. We have already determined chunks 2 and 3, so we are able to XOR 2 from this packet, but are still left with 1 and 4, and so must move on.
4. With the arrival of the fourth packet, we are able to resolve everything: data chunks 2 and 3 are already determined, and so we are able to XOR chunk 3 ($X_3 = 1$) with this new data packet (1) to get the value of the chunk 4,

$X_4 = 0$. With this new information, we are able to resolve X_1, as packet 3 gave us the equation

$1 = X_1 \oplus X_2 \oplus X_4 = X_1 \oplus 1 \oplus 0$. We can solve this to get $X_1 = 0$.

5. We have now solved for all the data chunks, with $X_1 = 0, X_2 = 1, X_3 = 1, X_4 = 0$.

As you might be able to tell, by choosing a good degree distribution for d, even when random incoming packets were lost (not shown), you were still able to recover all 4 symbols only from 4 received packets, despite the sender not knowing what packets you lost through the BEC.

<font color = blue> $\mathcal{Q}$uestion 1. Code

We've provided you with some starter code, including a Packet class, a Transmitter class, a Channel class, and a Receiver class. Your job is to complete the receive_packet() function in the Receiver class. Feel free to write any additional functions that you may need.

1) Packet Class & Utility functions

A packet consists of...

['chunk_indices', 'data']

chunk_indices: Which chunks are chosen

data: The 'XOR'ed data

In [2]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import json
import random

class Packet:
    size_of_packet = 256
    def __init__(self, chunks, chunk_indices):
        self.data = self.xor(chunks)
        self.chunk_indices = chunk_indices

    def xor(self, chunks):
        tmp = np.zeros(Packet.size_of_packet, 'uint8')
        for each_chunk in chunks:
            tmp = np.bitwise_xor(tmp, each_chunk)
        return tmp

    def num_of_chunks(self):
        return len(self.chunk_indices)
```

2) Transmitter & Encoder Class

You can initiate an encoder with a string! Then, `generate_packet()` will return a randomly encoded packet.

In [343]:
```
class Transmitter:
    def __init__(self, chunks, channel, degree_distribution):
        self.chunks = chunks
        self.num_chunks = len(chunks)
```

```python
        self.channel = channel
        self.degree_distribution = degree_distribution
        self.packets_sent = 0
        self.singles = 0
        self.current_split = 2
        self.degree = 2

    def generate_new_packet(self):
        if self.degree_distribution == 'single':
            #Always give a degree of 1
            n_of_chunks = 1
            self.singles += 1
        elif self.degree_distribution == 'double':
            #Always give a degree of 2
            n_of_chunks = 2
        elif self.degree_distribution == 'mixed':
            #Give a degree of 1 half the time, 2 the other half
            if random.random() < 0.5:
                n_of_chunks = 1
            else:
                n_of_chunks = 2
        elif self.degree_distribution == 'baseline':
            """
            Randomly assign a degree from between 1 and 5.
            If num_chunks < 5, randomly assign a degree from
            between 1 and num_chunks
            """
            n_of_chunks = random.randint(1,min(5, self.num_chunks))
        elif self.degree_distribution == 'sd':
            #Soliton distribution
            tmp = random.random()
            n_of_chunks = -1
            for i in range(2, self.num_chunks + 1):
                if tmp > 1/np.double(i):
                    n_of_chunks = int(np.ceil(1/tmp))
                    break
            if n_of_chunks == -1:
                self.singles += 1
                n_of_chunks = 1
        elif self.degree_distribution == 'sd_modified':
            values = np.arange(2, self.num_chunks + 1)
            n_of_chunks = -1

            if self.packets_sent / 1280 >= 1:
                offset = self.num_chunks
            else:
                for i in range(1, self.num_chunks + 1):
                    if self.packets_sent / 1280 < i * 1280 / (i + 1):
                        offset = i
                        break

            if offset == 1:
                n_of_chunks = 1
            else:
                tmp = random.random()
                for i in range(2, self.num_chunks + 1):
                    if tmp > 1 / np.double(i):
                        n_of_chunks = values[((i - 2) + (offset - 2)) % len(values)]
                        break
```

```
                if n_of_chunks == -1:
                    n_of_chunks = 1

        chunk_indices = random.sample(range(self.num_chunks), n_of_chunks)
        chunks = [ self.chunks[x] for x in chunk_indices ]
        return Packet( chunks, chunk_indices )

    def transmit_one_packet(self):
        self.degree_distribution = 'single' if self.singles < 1280 and self.packets_sent
% 8 == 0 else 'sd'
        packet = self.generate_new_packet()
        self.packets_sent += 1
        self.channel.enqueue( packet )
```

3) Channel Class

Channel class takes a packet and erase it with probability eps.

In [4]:
```
class Channel:
    def __init__(self, eps):
        self.eps = eps
        self.current_packet = None

    def enqueue(self, packet):
        if random.random() < self.eps:
            self.current_packet = None
        else:
            self.current_packet = packet

    def dequeue(self):
        return self.current_packet
```

4) Receiver & Decoder Class

You can initiate a decoder with the total number of chunks. Then, `add_packet()` will add a received packet to the decoder.

In [45]:
```
class Receiver:
    def __init__(self, num_chunks, channel):
        self.num_chunks = num_chunks

        # List of packets to process.
        self.received_packets = []

        # List of decoded chunks, where self.chunks[i] is the original chunk x_i.
        self.chunks = np.zeros((num_chunks, Packet.size_of_packet),dtype=np.uint8)

        # Boolean array to keep track of which packets have been found, where self.found
[i] indicates
        # if x_i has been found.
        self.found = [ False for x in range(self.num_chunks) ]
        self.channel = channel

    def receive_packet(self):
        packet = self.channel.dequeue()
        if packet is not None:
            if packet.num_of_chunks() == 1:
                self.update_chunks(packet, packet.chunk_indices[0])
            else:
```

```python
                self.peel(packet, True)

    def peel(self, packet, new=False):
        peeled_indices = []
        for chunk_index in packet.chunk_indices:
            if self.found[chunk_index]:
                packet.data = np.bitwise_xor(packet.data, self.chunks[chunk_index])
                peeled_indices.append(chunk_index)

        for chunk_index in peeled_indices:
            packet.chunk_indices.remove(chunk_index)

        if packet.num_of_chunks() > 1 and new:
            self.received_packets.append(packet)

    def update_chunks(self, packet, index):
        self.chunks[index] = packet.data
        prev = self.found[index]
        self.found[index] = True

        if not prev:
            self.update_received_packets()

    def update_received_packets(self):
        resolved = []

        for packet in self.received_packets:
            self.peel(packet)

            if packet.num_of_chunks() == 1:
                resolved.append(packet)

        for packet in resolved:
            self.received_packets.remove(packet)

        for packet in resolved:
            self.update_chunks(packet, packet.chunk_indices[0])

    def isDone(self):
        return self.chunksDone() == self.num_chunks

    def chunksDone(self):
        return sum(self.found)
```

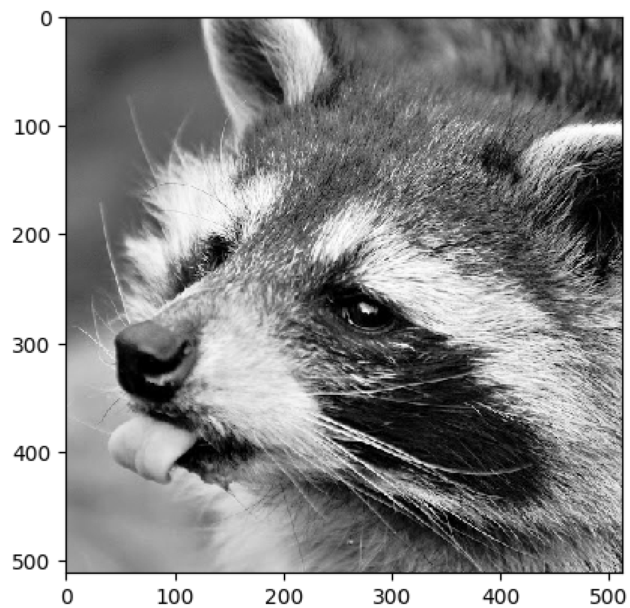
<font color = blue> $\mathcal{Q}$uestion 2. Sending the raccoon

In [46]:

```
from scipy import misc
import matplotlib.cm as cm

plt.style.use('default')
# pip3 install pillow
from PIL import Image
import numpy as np
l = np.asarray(plt.imread("raccoon.jpg"))
#converts the image to grayscale
x = np.zeros((512,512),dtype=np.uint8)
for i in range(512):
    for j in range(512):
        x[i][j] = l[i][j][0]*0.299+l[i][j][1]*0.587+l[i][j][2]*0.113

plt.imshow(x, cmap = cm.Greys_r)
```

Out[46]:

```
<matplotlib.image.AxesImage at 0x11097d2e8>
```



<font color = blue> a. Break up the image shown below into 1024 chunks of size 256 each.

In [47]:

```
tt= x.reshape(1,512*512)[0]
size_of_packet = 256
num_of_packets = 1024
assert len(tt) == size_of_packet * num_of_packets

#YOUR CODE HERE
chunks = []
for i in range(0, len(tt), size_of_packet):
    chunks.append(tt[i:i + size_of_packet])
```

<font color = blue> b. Here's a function that simulates the transmission of data across the channel. It returns a tuple containing the total number of packets sent, the intermediate image every 512 packets and the final image, and the number of chunks decoded every 64 packets). You'll use it next question.

In [48]:

```
#Returns a tuple (packets sent, intermediate image every 512 packets + final image, chunk
s decoded every 64 packets)
def send(tx, rx):
    num_sent = 0
    images = []
    chunks_decoded = []
    while not rx.isDone():
        tx.transmit_one_packet()
        rx.receive_packet()
        if num_sent % 512 == 0:
            images.append(np.array(rx.chunks.reshape((512,512))))
        if num_sent % 64 == 0:
            chunks_decoded.append(rx.chunksDone())
        num_sent += 1
    chunks_decoded.append(rx.chunksDone())
    images.append(rx.chunks.reshape((512,512)))
    return (num_sent, images, chunks_decoded)
```

<font color = blue> c. Using the 'single' degree distribution defined in the Transmitter class, send the raccoon over a channel with erasure probability 0.2. How many packets did you need to send? Display the data you receive every 512 packets in addition to the data you receive at the end.

You may find the following function useful:

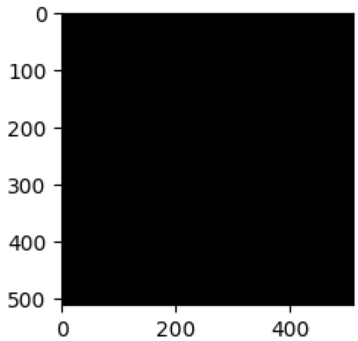
In [49]:

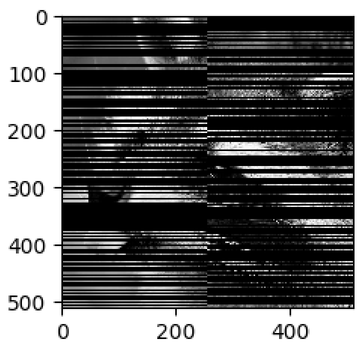
```
def visualize(image):
    #visualize takes in a 512 x 512 image. Therefore, you can't just pass in the chunks,
 which are 1024 x 256.
    plt.imshow(image, cmap = cm.Greys_r)
```

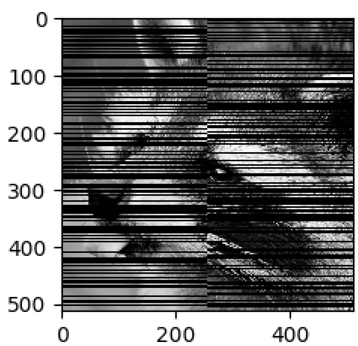
In [51]:

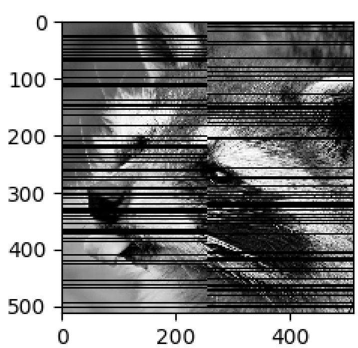
```
#YOUR CODE HERE
#you'll need to change the values
eps = .2
ch = Channel(eps)
tx = Transmitter(chunks, ch, 'single')
rx = Receiver(num_of_packets, ch)

#YOUR CODE HERE
num_sent, images, chunks_decoded = send(tx, rx)

print("The number of packets received: {}".format(num_sent))

### Show the intermediate data and the final picture
n_of_figures = len(images) #YOUR CODE HERE
fig = plt.figure( figsize=(8, 3*n_of_figures) )

for i in range(n_of_figures):
    fig.add_subplot(n_of_figures,1,i+1)
    visualize(images[i])
```

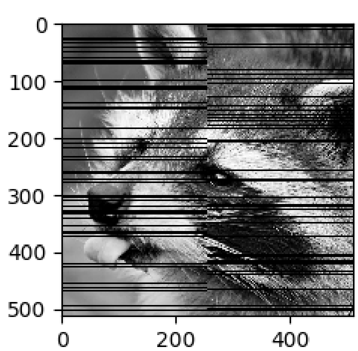
```
The number of packets received: 9306
```


0
100
200
300
400
500
0
200
400

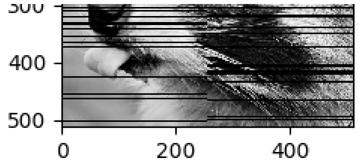

0
100
200
300
400
500
0
200
400

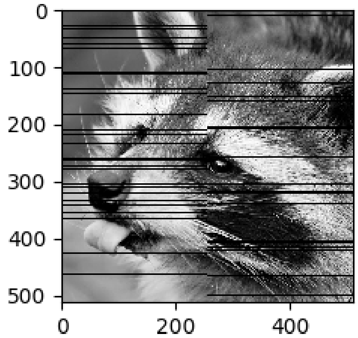

0
100
200
300
400
500
0
200
400

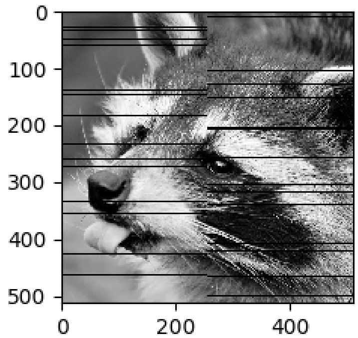

0
100
200
300
400
500
0
200
400

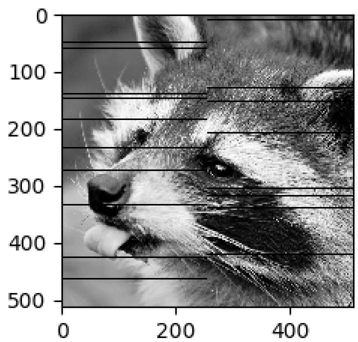

0
100
200
300
400
500
0
200
400

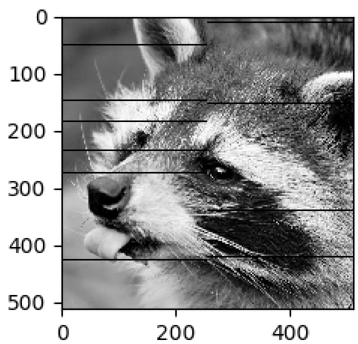

0
100
200

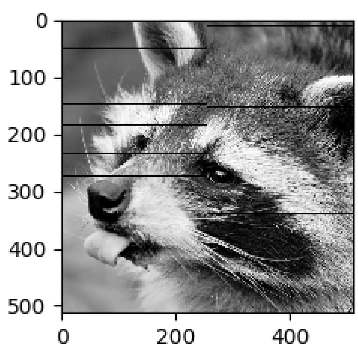

300
400
500
0
200
400

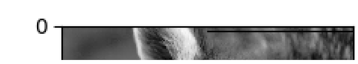

0
100
200
300
400
500
0
200
400

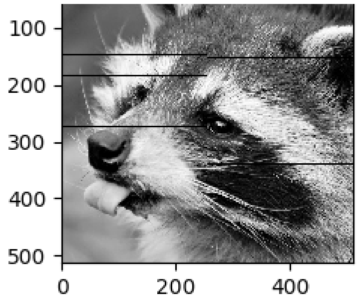

0
100
200
300
400
500
0
200
400

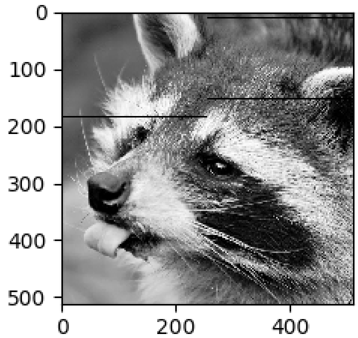

0
100
200
300
400
500
0
200
400

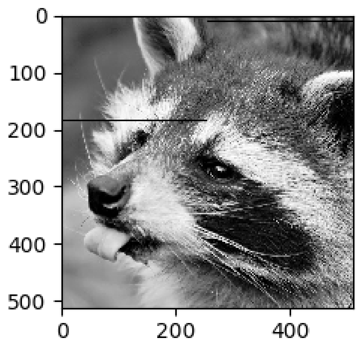

0
100
200
300
400
500
0
200
400

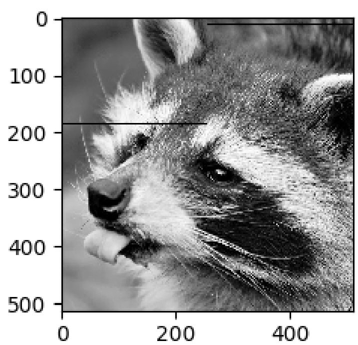

0
100
200
300
400
500
0
200
400

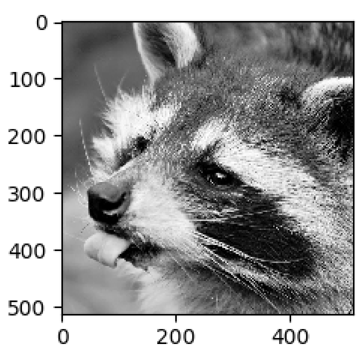

0

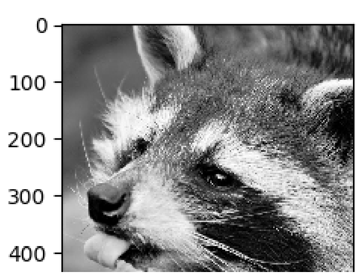

100
200
300
400
500
0
200
400

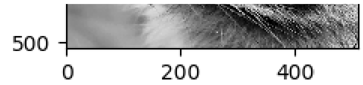

0
100
200
300
400
500
0
200
400

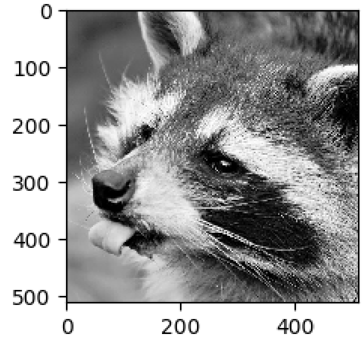

0
100
200
300
400
500
0
200
400

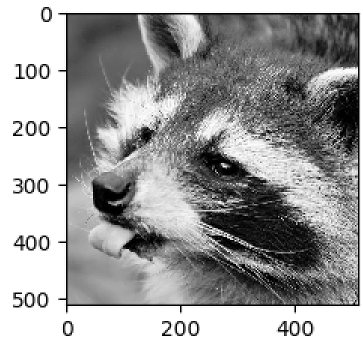

0
100
200
300
400
500
0
200
400

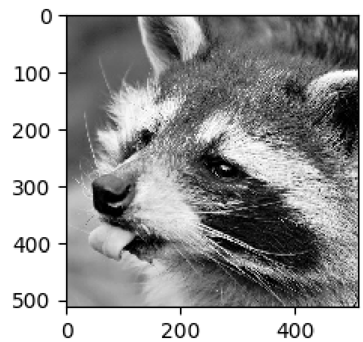

0
100
200
300
400
500
0
200
400


0
100
200
300
400









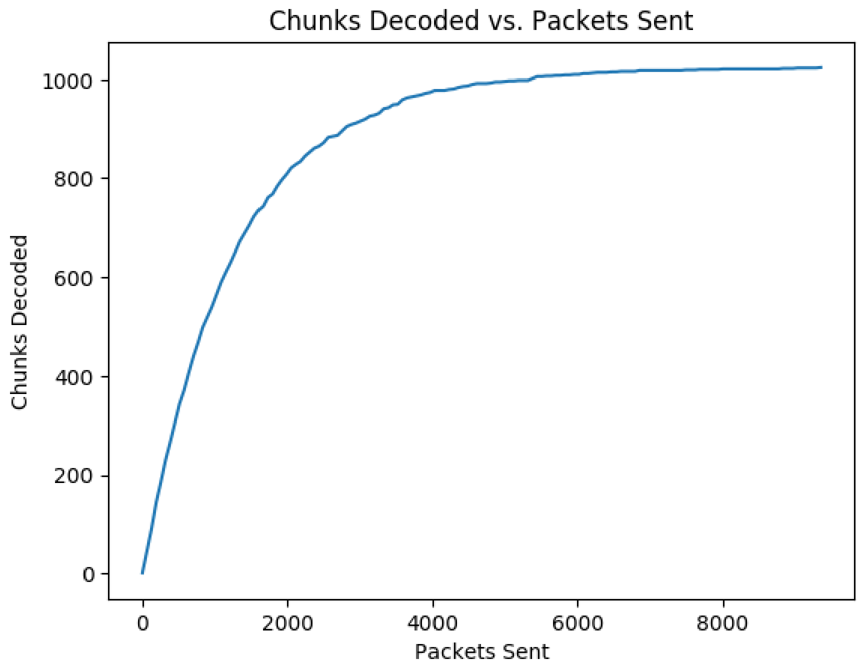
<font color = blue>d. Plot the number of chunks decoded as a function of the number of packets you send. (The chunks_decoded array should be helpful here)

In [69]:

```
#Plot the number of chunks decoded against the number of packets sent
plt.plot(np.arange(0, 64 * len(chunks_decoded), 64), chunks_decoded)
plt.xlabel("Packets Sent")
plt.ylabel("Chunks Decoded")
plt.title("Chunks Decoded vs. Packets Sent")
```

Out[69]:

```
Text(0.5,1,'Chunks Decoded vs. Packets Sent')
```



<font color = blue> e. Looking at the graph, we see that it gets harder and harder to find the rest as we decode more and more chunks. Does this remind you of a well known theoretical problem?

Hint: Try out some small examples!

Coupon collector's problem

<font color = blue> f. Using the 'double' degree distribution defined in the Transmitter class, send the raccoon over a channel with erasure probability 0.2. Don't worry about intermediate plots this time. What happens?

In [344]:

```
#YOUR CODE HERE
eps = .2
ch = Channel(eps)
tx = Transmitter(chunks, ch, 'double')
rx = Receiver(num_of_packets, ch)

#YOUR CODE HERE
num_sent, images, chunks_decoded = send(tx, rx)
print("The number of packets received: {}".format(num_sent))
```

```
The number of packets received: 8753
```

None of the packets can be decoded since they are all degree 2, with no degree 1 packets.

<font color = blue> $\mathcal{Q}$uestion 3. Randomized Distributions¶

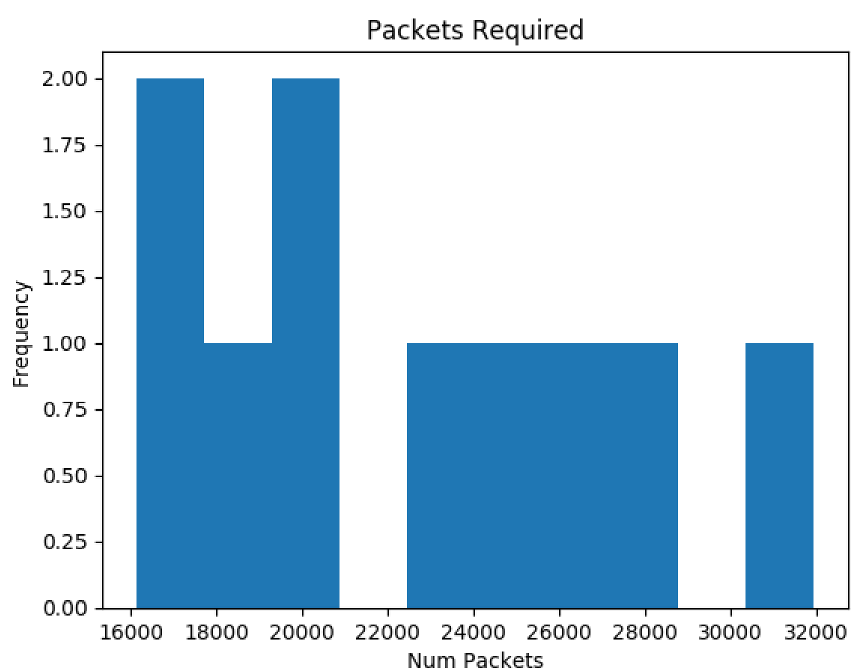
<font color = blue> a. You have seen two degree distributions so far. Both of these have been deterministic, and one worked better than the other. Let's try a different degree distribution. Using the 'baseline' degree distribution, send the raccoon over a channel with erasure probability 0.2 over multiple trials. For each trial, record the number of packets sent for the image to be decoded.

In [76]:

```
num_trials = 10
#YOUR CODE HERE
eps = .2
ch = Channel(eps)
tx = Transmitter(chunks, ch, 'baseline')

packets_required = []

for _ in range(num_trials):
    rx = Receiver(num_of_packets, ch)
    num_sent, images, chunks_decoded = send(tx, rx)
    packets_required.append(num_sent)
    #YOUR CODE HERE

#Plot the packets required as a histogram
plt.figure()
plt.hist(packets_required)
plt.title("Packets Required")
plt.xlabel("Num Packets")
plt.ylabel("Frequency")
```

Out[76]:

```
Text(0,0.5,'Frequency')
```



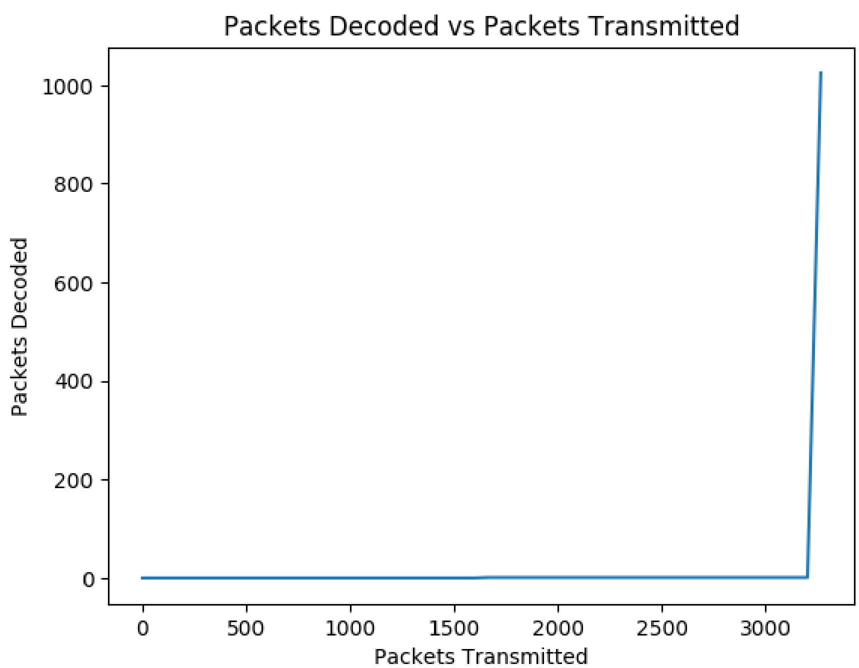
<font color = blue> b. Let's examine one final degree distribution. Using the 'sd' degree distribution, send the image over a channel with erasure probability 0.2. Plot the number of packets decoded against the number of packets transmitted.

In [95]:

```
#YOUR CODE HERE
eps = .2
ch = Channel(eps)
tx = Transmitter(chunks, ch, 'sd')
rx = Receiver(num_of_packets, ch)

#YOUR CODE HERE
num_sent, images, chunks_decoded = send(tx, rx)
plt.figure()
plt.plot(np.arange(0, len(chunks_decoded) * 64, 64), chunks_decoded)
plt.xlabel("Packets Transmitted")
plt.ylabel("Packets Decoded")
plt.title("Packets Decoded vs Packets Transmitted")
```

Out[95]:

```
Text(0.5,1,'Packets Decoded vs Packets Transmitted')
```



<font color = blue> $\mathcal{C}$ompetition

Alice has just finished eating dinner, and with her EE 126 homework completed early for once, she plans to sit down for a movie night (she wants to make use of the 30-day free trial of Netflix!). While Alice is surfing Netflix she decides she wants to stream Interstellar. Alice's laptop drops packets with $p = 0.2$. You, the Chief Technology Officer of Netflix, know that given the heavy workload of EE 126, this may be your only chance to convert this freeloading customer into a permanent one, but to do so you're going to have to make sure her viewing experience is perfect.

Concrete specs:

- You are given an erasure channel with drop probability $p = 0.2$.
- You must define a degree distribution (which can vary as a function of the # of transmissions already sent) to minimize the number of total packets needed to be sent for the raccoon to be decoded. Run your code for 10 trials to get a good estimate of the true number of transmissions needed per image while they watch their movies. Each trial, your score is

$$\frac{\text{\# of packets successfully decoded from the first 512 packets}}{512} + \frac{\text{\# of packets successfully decoded from the first 1024 packets}}{1024}.$$
$$+ \left\lfloor \frac{\text{\# of packets successfully decoded from the first 2048 packets}}{1024} \right\rfloor$$
$$+ \left\lfloor \frac{\text{\# of packets successfully decoded from the first 4096 packets}}{1024} \right\rfloor$$
$$+ \left\lfloor \frac{\text{\# of packets successfully decoded from the first 6144 packets}}{1024} \right\rfloor$$

- Note the floor function in the later stages - you can only get the point if you fully decode the file with the alloted number of packets
- **You may work in teams of up to three.**
- One thing you can do is add a packets sent argument with a default argument None to generate and transmit in Transmitter

Good luck!

If you place in the top 3 in the class you will be awarded bonus points and full credit for the homework, as well as get to present your strategy to the entire course staff!

Besides the top 3 submissions:

Any score above 3 will receive full credit. Everyone who scores above 3 points will receive bonus credit that is proportional to their score!

In [193]:

```
from math import floor

def score(chunks_decoded):
    c_d = chunks_decoded
    s = c_d[8]/512+c_d[16]/1024
    arr = [33,65,97]
    for i in arr:
        if i >= len(c_d):
            s += 1
    return s
```

In [360]:

```python
eps = .2
ch = Channel(eps)

packets_required = []
chunks_decoded_trials = []

for _ in range(num_trials):
    tx = Transmitter(chunks, ch, 'sd_modified')
    rx = Receiver(num_of_packets, ch)
    num_sent, images, chunks_decoded = send(tx, rx)
    packets_required.append(num_sent)
    chunks_decoded_trials.append(chunks_decoded)
```

In [361]:

```python
sum(map(score, chunks_decoded_trials)) / num_trials
```

Out[361]:

```
3.0796875
```

In [362]:

```python
list(map(score, chunks_decoded_trials))
```

Out[362]:

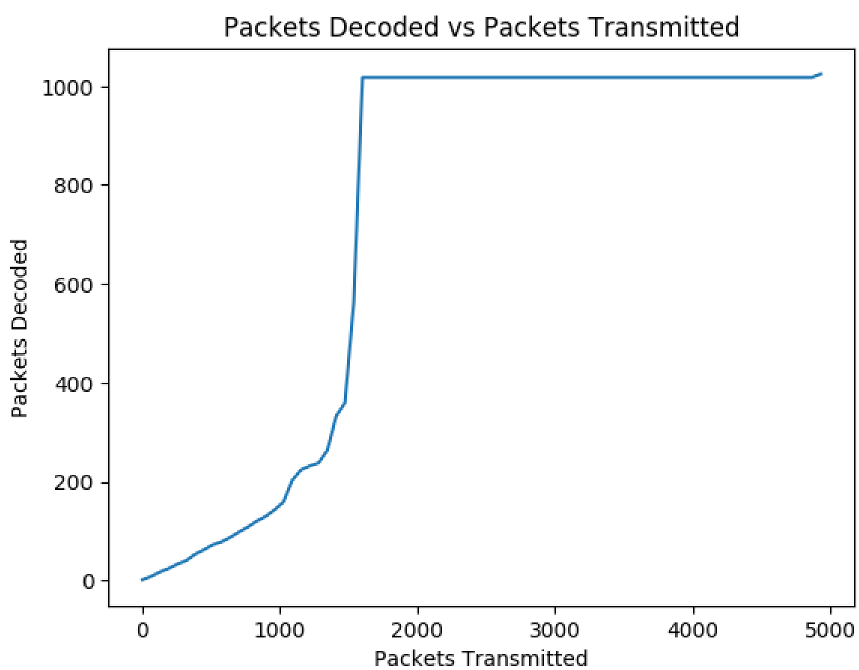
```
[3.251953125,
 3.25390625,
 3.2880859375,
 3.2939453125,
 3.306640625,
 3.2255859375,
 3.2890625,
 1.279296875,
 3.33984375,
 3.2685546875]
```

In [348]:

```python
plt.figure()
plt.plot(np.arange(0, len(chunks_decoded_trials[4]) * 64, 64), chunks_decoded_trials[4])
plt.xlabel("Packets Transmitted")
plt.ylabel("Packets Decoded")
plt.title("Packets Decoded vs Packets Transmitted")
```

Out[348]:

```
Text(0.5,1,'Packets Decoded vs Packets Transmitted')
```



<font color = blue> $\mathcal{R}$esults

Report the average score (averaged over 10 trials):

Average Score: $\boxed{3.0796875}$

<font color = blue> $\mathcal{S}$ummary

Answer the following in 1-2 paragraphs (this should be answered individually):

- Who were your teammates?
- What did you learn?
- What is the basic inuition behind your final strategy?
- How did your strategy evolve from your first attempt (what worked and what failed)?
- How would your strategy change if the value of p of the BEC was not known?

<font color = red> I worked individually. I learned how the probability mass function of the degree should be varied according to the packets sent out. The intuition behind my final strategy was to send out more lower degree packets first, then more higher degree packets later on. My first attempt involved a mix of the single and soliton distributions, but this was still suboptimal as it did not have enough of a variation in packet degrees over time. If p was not known, my strategy would have had to assume a default $p = .5$ for most conservative estimate in determining the expected number of packets to send out.

References

[1] D. Mackay. Information Theory, Inference, and Learning Algorithms. 2003
[2] http://blog.notdot.net/2012/01/Damn-Cool-Algorithms-Fountain-Codes (http://blog.notdot.net/2012/01/Damn-Cool-Algorithms-Fountain-Codes)