Reliability is important when computers are transmitting information. However, in the real world, transmission errors occur. These are typically caused by one of three sources:

- Interference. EM radiation causes electronic noise when it strikes metal wires; this creates interference noise that makes it harder for the signal to be "heard".
- Distortion. Metal objects can block radio waves, making them difficult or impossible to hear.
- Attenuation. Signals naturally become weaker with distance.

Errors can never be eliminated completely, but they CAN be detected and corrected. This adds overhead, but will increase clarity. This means that the transmission will require more time to send or more time to decipher, but will also come through more clearly.

Some types of error are more important than others. If one bit in a bank transfer is changed:
00001010 -> 10001010
The transfer changes from $10 to $138! This is an important error.
However, one single bit changed in a picture is much less important. In fact, under most circumstances, a viewer would not be able to detect any change in the picture itself.

There are a few types of interference.

- A single bit error occurs during very short interference; one single bit is changed.
- A burst error is a large segment of malformed information. This usually occurs during longer durations of interference. The "burst length" is the length of the malformed information; it is measured from the first malformed bit to the last, and includes any correct bits in between. This means that if three bits changed and a fourth one is in the middle, all four bits are considered part of the burst length.
- Erasure or ambiguity occurs when a signal is not clearly a 1 or a 0. Sometimes, interference can cause bits to come across at ambiguous voltages; if a "1" is 10 volts and a "0" is 0 volts, a 4 would be ambiguous.

There are two ways to handle errors. Channel coding, the techniques for overcoming errors, can be divided into two categories.

- Forward Error Correction (FEC) adds additional data, like a checksum, to the transmission itself. This allows the receiver to understand if the data came across clearly.
- Automatic Repeat reQuest (ARQ) has a sender and receiver in constant communication. The sender and the receiver exchange information to make sure that all of the information arrived correctly. TCP uses ARQ; we will discuss this more when we cover TCP in detail.

For single parity checking, the sender and receiver send an extra bit, called a parity bit. The parity bit counts the number of "1" bits in the transmission. "Even parity" makes the parity bit a 1 if there are an odd number of 1s, and "odd parity" makes the parity bit a 0 if there are an odd number of 1s.
Single parity checking has very low overhead, as it sends only one single extra bit. However, it has two downsides: first, single parity can only detect errors; it cannot change them. Additionally, single parity can only detect errors if an odd number of bits were changed. If an even number of bits were changed during burst interference, single parity cannot correctly detect the error.

The strength of an error detection system is determined by its Hamming distance. The Hamming distance is the number of bits that must be changed before the message falsely passes a security check. For example, if we use odd parity on "01", it becomes "010". If just two bits are changed, from "010" to

“100”, the message will still be interpreted as correct. Parity therefore has a very short Hamming length, and is not particularly good at detecting errors.

Now consider a more secure mode of transmission: Row and Column (RAC) Parity. RAC parity converts a stream of data into a matrix, and then computes the parity of each row and each column. A 16 bot stream would become a 3x4 matrix. RAC parity allows the row and column of a bit change to be detected; it can therefore not only detect the presence of an error, but its location, too. However, it can only detect single-bit errors.

A 16 bit checksum algorithm works its way over the entire message, in order to check to see if it is correct. The algorithm divides the message up into units that are 16 bits long, and then adds all of these together. Any overflow is added to the checksum in its least significant bit. The result is then inverted via one’s compliment. When the receiver gets its message, it performs the same function, but it also adds the checksum. If it gets a result of 0, there are no errors.

```
 0010 1000 0110 0101
 0110 1100 0110 1100
+0110 1111 0010 0001
-----------------------------
1 0010 0011 1111 0010
^ overflow is taken and added to the LSB of the checksum.

 0010 0011 1111 0010
+                    1
-----------------------------
 0010 0011 1111 0011
 ^^^^ ^^^^ ^^^^ ^^^^ this is then inverted through one’s compliment to get the checksum:

 1101 1100 0000 1100
 ^^^^ ^^^^ ^^^^ ^^^^ this is our checksum. If we take the original addition function and add this to it,
we should get a zero.

 0010 1000 0110 0101
 0010 1100 0110 1100
 0110 1111 0010 0001
+1101 1100 0000 1100
-----------------------------
1 1111 1111 1111 1110
^ this is added to the end of the checksum.

 1111 1111 1111 1110
+                    1
-----------------------------
 1111 1111 1111 1111
 ^^^^ ^^^^ ^^^^ ^^^^ this is inverted by one’s compliment

 0000 0000 0000 0000
 ^^^^ ^^^^ ^^^^ ^^^^ this equals zero. This means that there were no errors in transmission.
```

Cyclic Redundancy Code (CRC) is used with Ethernet or other high-speed networks. There are three key properties of CRC:

- It can transmit and check a message of arbitrary length.
- The error detection works very well.
- It is based on sophisticated math, but can be executed extremely quickly by hardware.

Binary numbers are divided under the assumption of no carries. We can think of it as subtraction being replaced by an XOR. Suppose our message is "1010" and our CRC constant is "1011". The division looks like this:

1011 / 1010 000
^^^ these are added because a 4-bit divisor yields a 3 bit CRC, a 3 bit divisor yields a 2-bit CRC, etc.

1010 000
/1011
0010
We divided it by itself, so the first digit of the divisor is 1.
001 000
/ 000 0
01 00
We did not divide by anything, so the second digit of the divisor is 0.
01 000
/ 0 000
1 000
We did not divide by anything, so the third digit of the divisor is 0.
1 000
/ 1 001
011
The remainder is "001" – this is our CRC.

CRC is complicated, and is further explained by this article:
https://en.wikipedia.org/wiki/Cyclic_redundancy_check#Computation
Be sure to review the article; it will add clarity to today's lecture.