Mobile Systems

Todd Davies

May 30, 2015

Introduction

Now that the mobile telephone has evolved into a powerful computer, the mobile dimension of computing is a vital part of Computer Science. This unit will give insights into many issues of mobile systems, including wireless communication networks, the processing of speech, music and other real-time signals, the control of bit-errors and maximising battery life. The techniques and software which underlie commonplace applications of mobile computing systems, including smart-phones, tablets, laptop computers, MP3 players and GPS satellite navigation, will be addressed.

Aims

Computing is becoming increasingly mobile. This unit will give insights into the issues of mobile systems, covering mobile communications, real-time signals such as speech, video and music, codecs, and maximising battery life.

- Commonplace examples of mobile computing systems: - mobile phones; - MP3 players; - laptop computers; - PDAs; - GPS satellite navigation.
- Real-time signals
- Analogue and digital signals; - time and frequency domain representations; - sampling, aliasing, quantization; - companding; - real-time computation.
- Coding, decoding and compression
- GSM speech coding; - MP3 music, JPEG image and MPEG video coding & decoding; - error correcting codes; - communications coding schemes.
- Mobile communication
- Transmitting real-time information over wireless networks; - principles of cellular and ad-hoc networks; - Coding of multimedia signals - to increase the capacity of radio channels; - to minimise the effect of transmission errors.
- Maximising battery life
- May be addressed at many levels including: - chip design; - signal coding and processing; - medium access control; - transmit power control.

Contributing

These notes are open-sourced on Github at `https://github.com/Todd-Davies/second-year-notes`. Please feel free to submit a pull request if you want to make any changes, or maybe open an issue if you find an error! Feedback is very welcome; you can email me at todd434@gmail.com.

Contents

1 Course intro **3**
1.1 What's in a smartphone? 3

2 Signals in mobile systems **3**
2.1 Generating waves 3
2.2 Sampling waves 3
2.2.1 The Sampling Theorem 4
2.2.2 Aliasing 4
2.3 Decibels 4

3 Frequency Domain Processing **5**
3.1 Sinusoids 5
3.2 Fourier Series 5
3.3 Discrete Fourier Transform (iDFT) 7

4 Encoding and storing signals **7**
4.1 Bitrates in different settings 7
4.2 Quantisation 8
4.2.1 Quantisation error 9
4.3 Differential encoding 10
4.4 Linear Predictive Speech Coding 11
4.4.1 LPC-10 11
4.4.2 Codebook Excited LPC 11
4.5 Comfort noise 12
4.6 Error correction, detection & prevention . 12
4.6.1 Forward Error Correction 12
4.6.2 CRC checks 13
4.6.3 Convolutional Coders 14
4.6.4 Miscellaneous FEC stuff 14
4.7 Image Coding 14
4.7.1 Frequency domain processing of images 15
4.7.2 The JPEG standard 15
4.7.3 Huffman Coding 15
4.7.4 Encoding Movies 17

5 Mobile Networks **17**
5.1 Cellular networks 18
5.1.1 The evolution of cellular technologies 18
5.1.2 2G in detail 19
5.1.3 3G in detail 19
5.1.4 The details of 4G 21
5.1.5 5G and the future! 21
5.2 GPS 22
5.3 Multimedia communications 22
5.3.1 Streaming protocols 23
5.3.2 VoIP Telephony 23
5.3.3 Optimising power 24

6 Medium Access Control **24**
6.1 ALOHA 25
6.1.1 Slotted ALOHA 25
6.2 Carrier Sense Multiple Access 25
6.2.1 CSMA collision detection 26
6.3 Wireless contention 26
6.3.1 Wireless problems 27
6.3.2 Bluetooth 27

7 Real time computation **27**
7.1 Scheduling 28
7.2 External events 28
7.3 Buffering 28
7.4 Timing 28
7.5 Real time communications/Operating Systems 29

8 Chip Design **29**
8.1 Jevon's paradox 30
8.2 Low power design 30
8.2.1 CMOS 30

9 Code **32**

1 Course intro

A smartphone is a mobile phone running a mobile operating system, with advanced capabilities with regard to computing power and connectivity. They are much more advanced than traditional feature phones, and have lots of features, including cameras, multimedia functionality, GPS, touch screens etc.

There are three main mobile operating systems in use today:

Android
Founded in 2003 by Andy Rubin and backed by Google. It's mostly free and open source, and holds a very strong position in the market.

iOS
Introduced in 2007 by Apple, this is a closed source operating system. The first iOS phones were very groundbreaking in terms of their technology, and were the first to feature touch screens, which are now ubiquitous in the market.

Windows Phone, Blackberry, Symbian, Palm OS etc could be listed here too.

1.1 What's in a smartphone?

Modern smartphones are jam packed with technology, containing cameras, multimedia, GPS, high resolution touch screens, motion sensors, bluetooth, RFID, NFC and even more stuff besides. They let you talk over multiple different networks (2G, 3G, 4G etc), and access data using the same methods.

Of course, the millions of apps available for them is also a massive attraction.

Smartphone Operating Systems need to be very advanced in order to step up to the tasks required of them by users. They need to be multitasking, to run lots of different apps, and interface with the typical hardware a normal computer might use (DMA, input devices etc). However, they also need to have real-time elements in, since they must be able to drive sound, IO, radio communications, digital signal processing etc

2 Signals in mobile systems

Signals such as speech and music arrive at the device as physical, analogue quantities that vary in a continuous manner over time. If we plot a graph of voltage over time, then we get a waveform for that signal. We can convert analogue signals to *discrete time signals* by sampling them at set intervals (discrete points in time). This produces a list of numbers from $-\infty$ to ∞.

2.1 Generating waves

In order to create a wave with a period of T seconds, you can use the formula:

$$y = sin(\frac{2\pi x}{T})$$

Of course, since $f = \frac{1}{T}$, the frequency of the wave is the reciprocal of the time period.

2.2 Sampling waves

We can work out the frequency of a wave, by finding how many complete cycles it undergoes in one thousand samples, and multiplying that by the sample rate. For example, if a wave has ten cycles from 1000 samples at $30,000Hz$, then its frequency will by $\frac{10}{1000} \times 30,000 = 300Hz$. This is visible in Figure 1.

Wav files are merely just lists of binary numbers that have been sampled and possibly quantised; it's a really simple format.

Not all waves are so easy to analyse, rarely will a wave be at just one frequency, instead, they are usually 'noisy' and will be composed of many waves added together. Many waves (that aren't just

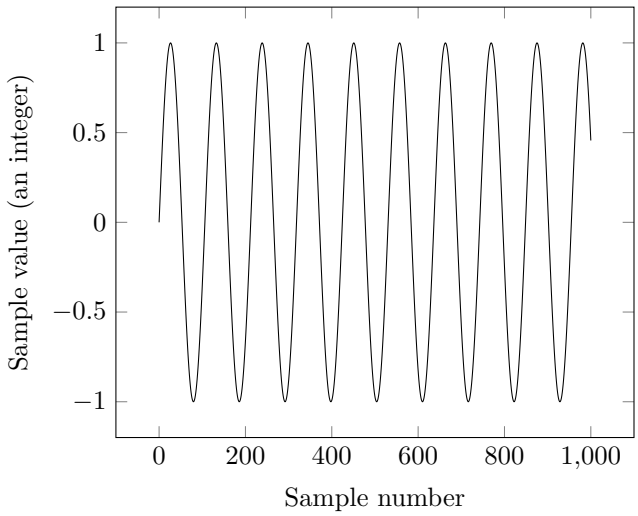


Figure 1: 1000 samples of an analogue sin wave of frequency $300Hz$ sampled at $30,000$Hz.

noise) will have a discernible frequency that you can extract. The 'extra' waves on top of this frequency aren't necessarily bad, they might add harmonics, or texture to a sound.

2.2.1 The Sampling Theorem

Also known as *Nyquist criterion*.

The Sampling Theorem states that if a signal has all of its spectral energy below BHz, and is sampled at FHz, where $F \geq 2B$, then it can be reconstructed *exactly* from the samples and nothing is lost.

In other words, if you are sampling a signal at *less than double* of the maximum frequency of the signal, then you won't be able to fully reconstruct the signal from the samples, and you will get distortion.

Since music is usually sampled at 44.1kHz, we can accurately sample music with frequencies up to around 22kHz. Since humans can only hear up to around 20kHz, and speech rarely has frequencies above 3.4kHz, so the sample rate for it can be much lower, as we will see later on in the course.

2.2.2 Aliasing

If we sample at less than twice the frequency of the wave, then we will get distortion, known as aliasing. Even if we don't want frequencies higher than half our sample rate, we need to filter them out anyway, since otherwise, we'll get aliasing when we sample them.

To be precise, if a wave of frequency f is sampled at a frequency of below $2f$ (lets call this F, then the sampled output will be a wave of frequency $F - f$.

For example, if we sample a 6kHz wave at a frequency of 10kHz, when we will get a wave of frequency $10 - 6 = 4$kHz.

This is bad, because if you have harmonics in a musical note that are higher than half the sampling frequency, then these harmonics will be out of tune post sampling, and will go down when it's supposed to go up.

It may be hard to work out exactly why aliasing occurs, but Figure 2, showing a 6kHz wave sampled at 8kHz makes it easier to see:

2.3 Decibels

Sound can be measured in decibels, which is a logarithmic ratio. The formula is as follows:

$$\text{dB} = 10 \times log_{10}\left(\frac{s_1}{s_2}\right)$$

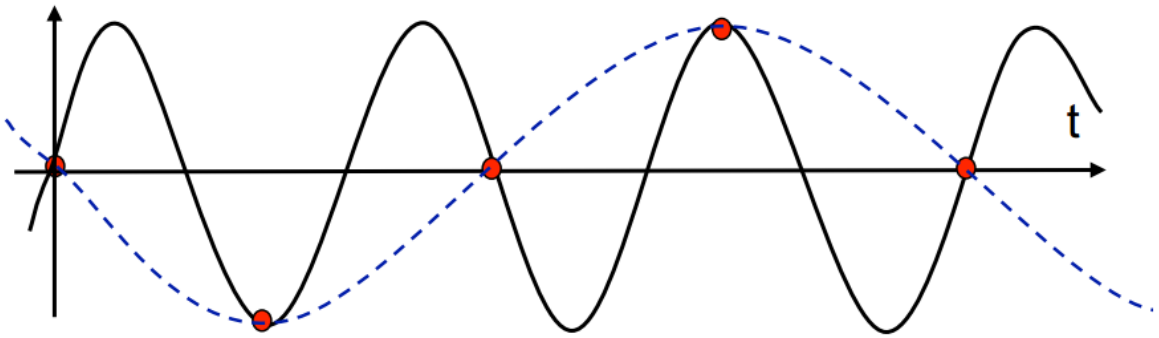


Figure 2: A 6kHz wave sampled at 8kHz has a post-sampling frequency of 2kHz

If s_1 is twice as loud as s_2, then it will be $10 \times \log_{10}(2) = 3$dB louder than s_2.

The following table shows the power ratio $(\frac{s_1}{s_2})$ against the equivalent decibels:

Power ratio	**Decibels**
$\frac{1}{2}$	-3
1	0
2	3
4	6
10	10
100	20
1000	30
10^5	50
10^{10}	100

3 Frequency Domain Processing

Instead of processing waves, and parts of waves, how about we turn them into frequencies and process those instead? Frequencies are a lot easier to deal with in many ways since they are easier to understand and smaller to store.

3.1 Sinusoids

A sinusoid is a sine wave delayed by D seconds, and is given by the formula:

$$y = M \times cos(2\pi F(t - D))$$

Figure 3 shows two sinusoids; one where $D = 30°$ and another where $D = 0$.

Figure 3 is a simple sine wave, but any wave (such as the one in Figure 4) that is periodic can be found to have a fundamental frequency of $\frac{1}{T}$, where T is the period of the wave. A recording of a voice, or musical instrument may be found to have a similar waveform to that of Figure 4 as long as we 'zoom in' to a segment short enough that any gradual change in frequency is negligible. This is called **psudo-periodicity**.

3.2 Fourier Series

Any periodic wave of frequency F can be written as:

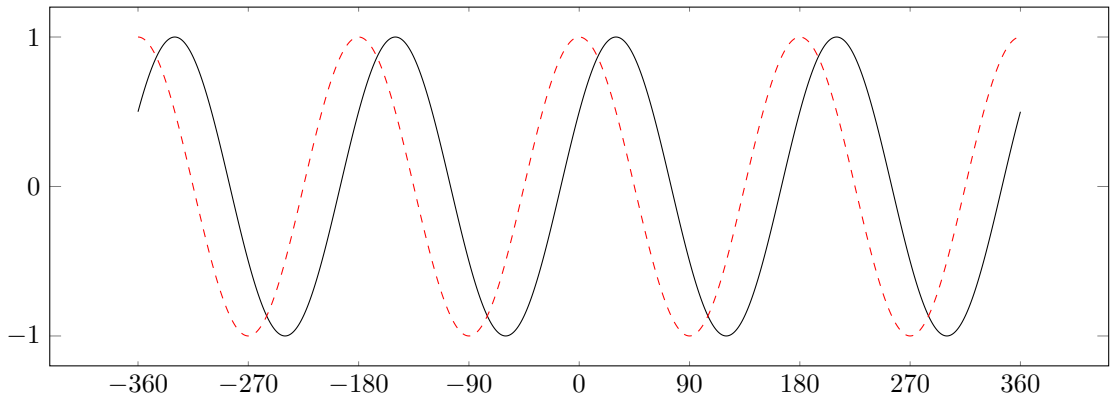


Figure 3: 1000 samples of a sinusoid where $D = 30°$ (black) and where $D = 0$ (dotted red).

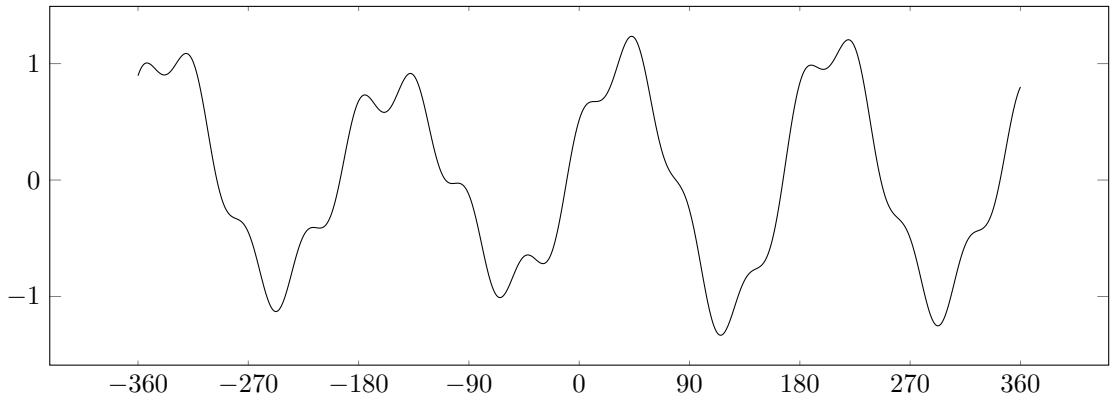


Figure 4: A complex wave with a frequency of $\frac{1}{180} = 0.00\bar{5}$Hz (if the x axis is in seconds).

$$\begin{aligned} x(t) = A_0/2 &+ A_1 cos(2\pi Ft) + B_1 sin(2\pi Ft) \\ &+ A_2 cos(2\pi 2Ft) + B_2 sin(2\pi 2Ft) \\ &+ A_3 cos(2\pi 3Ft) + B_3 sin(2\pi 3Ft) \\ &+ \dots \end{aligned}$$

This is sometimes called its 'cos and sin' form, but we could re-write it as:

$$x(t) = \frac{M_0}{2} + \sum_{k=1}^{\infty} M_k \cos(2\pi(kF)t + \phi_k)$$

Here, each time the summation iterates, we are adding the next harmonic of the sound. The fundamental frequency F is the first harmonic when $k = 1$. M_k is the magnitude, and ϕ_k is the frequency of the k^{th} harmonic.

Because the harmonics get higher and higher, we can't sample the whole series because otherwise the frequency of the harmonics will start to become higher than half the sampling frequency and we will get aliasing (see Section 2). In that case, we only go up to $k = \frac{N}{2} - 1$, where $F(\frac{N}{2} - 1) \leq \frac{Fs}{2}$. If we took Fs samples every second, then the formula would be:

$$x(n) = \frac{M_0}{2} + \sum_{k=1}^{\frac{N}{2}-1} M_k \cos\left(2\pi(kF)\frac{n}{Fs} + \phi_k\right)$$
$$\text{Where } n = 0, 1, \dots, \text{size}(x)$$

The more harmonics we compute, the closer we can get to reconstructing the wave from them afterwards.

3.3 Discrete Fourier Transform (iDFT)

The DFT converts samples of length n into $n/2 - 1$ samples and a 'dc term'. They are arranged in a length n array, where the second half is the same as the first half but reversed. The 0^{th} element of the array is the DC term, but the i^{th} element is the i^{th} harmonic.

The algorithm to convert from the time domain into the frequency domain is *N-point DFT*. To do the opposite, we can use the *N-point inverse DFT* algorithm.

iDFT is the abbreviation for the inverse cosine transform, and no, you can't find it in the Apple store.

DFT is used to:

- Perform spectral analysis on a signal to find out what components are present.
- Convert into the frequency domain before applying signal processing (and converting back again afterwards). For example, you could filter out noise from a sine wave, since you could remove all frequencies with an amplitude below a certain level.

4 Encoding and storing signals

There are a variety of different ways to encode sound. The two main types of coders for talking over the phone are *waveform* coders, and *parametric coders* (sometimes called vocoders).

Waveform coders
Waveform coders 'operate on' the sound waves directly, and aim to change the wave so that it is transmitted in an optimal way. They are simple to understand and implement, but can't achieve *really* low bit rates. They aim to preserve the exact shape of the wave.

Parametric coders
Parametric coders try and model human speech by exploiting how we produce sound with our vocal chords and mouth shape. They don't try and preserve the exact wave shape, but instead try and describe *perceptually significant features* as sets of parameters. This is more complicated and harder to implement, but achieves lower bit rates.

Techniques that fall into both types of coder will be discussed in this section.

Dynamic range is the ratio between the loudest sound we can hear, and the most quiet sound we can hear.

Humans can hear frequencies between 20Hz and 20kHz, and have a dynamic range of 120dB. We can represent six decibels per bit of information using uniform quantisation (see section 4.2), so we need about $\frac{120}{6} = 20$ bits to properly represent the sample.

4.1 Bitrates in different settings

Sometimes, 20 bits per sample is too much, so different bit rates are used in different applications:

CD's
Unfortunately, 20 bits per sample was too much for CD's (you wouldn't be able to fit enough songs on each CD), so instead, 16 bits per sample was adopted as the standard. In order to 'lose' these four bits, we make the quiet bits of the songs louder (since we probably wouldn't be able to hear them anyway).

As a consequence of this, a song on a CD plays at a rate of $16 \times 44100 \times 2 = 1411$kbit/s. The '2' comes from the fact that CD's store stereo music, and therefore have two channels; one for left and one for right.

Landlines
Landlines are band-limited to a range of frequencies from 50Hz to 3.4kHz. Obviously this is much, much less than the dynamic range that humans can hear, but we only usually talk in frequencies covered by the range provided. This means that the sound will lose its 'naturalness' but not intelligibility (supposedly).

In practice, sometimes, vowel sounds like 'f' and 's' can be mixed up.

Band limiting is when a sound is Fourier transformed, and frequencies above or below a certain frequency are stripped out.

The sample rate for telephone quality speech is 8kHz, with 8bit/sample. This gives a bitrate of 64kbit/s. In order to use just 8bit/sample, we need to use non-uniform quantisation (like in the lab), such as μ-law quantisation.

Mobile Telephones

Mobile phones use a bit rate that changes based on factors such as the signal quality, but has a minimum value of 4.75kbit/s, and a maximum value of 12.2kbit/s. AMR encoding (see Section 4.4.2) is used to achieve such low bitrates.

4.2 Quantisation

When we've sampled a wave, if we leave all of our readings as floating point numbers, then we could be using a lot of space to store each sample. We could slightly reduce the quality of our sound by *quantising* it. This is when you map the continuous values onto a range of integers, where the range of integers spans a power of two (so you can use as few bits as possible).

You can see that the third wave in Figure 5 has been quantised into five integers, since there are five discrete values in the waveform $(0, 1, 2, 3, 4)$ (requiring three bits), and there will be an extra bit to indicate the sign.

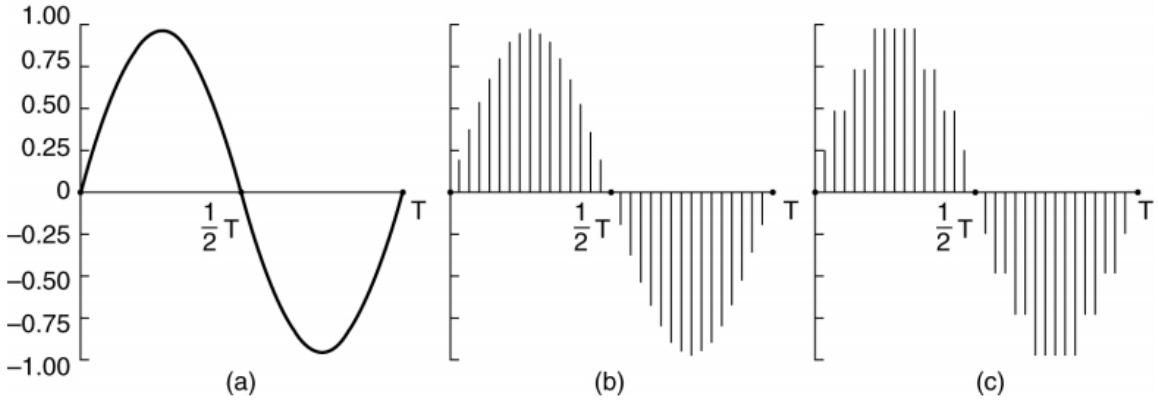


Figure 5: An analogue wave (a), after its sampled (b), and once it's been quantised (c)

Quantisation produces noise (known as quantisation noise), since errors are introduced in the process. If there are many bits per sample (e.g. 16), then this error will be small, but if you only used say, five bits, then the error would be noticeable. As a consequence of this, we have to work a trade-off between storage capacity or bandwidth and quality.

There are two types of quantisation; uniform and non-uniform:

Uniform quantisation

A simple type of quantisation, each binary number represents a voltage, where incrementing the binary representation will correspond to an increase in the voltage of ΔV, called the *quantisation step-size*.

It might be hard to decide on a value for Δ, because different sounds will span different ranges of amplitude as shown in Figure 6.

One solution to this, is to adjust the value of Δ on a sample-to-sample basis. This uses up extra bandwidth though, so is not a preferred solution.

Non-uniform quantisation

Non-uniform quantisation is when the quantisation step-size is not the same between different quantised values, as evidenced in Figure 7.

Non-uniform quantisation is implemented like so:

1. Apply accurate uniform quantisation.
2. Apply a 'companding' formula, such as μ-law.
3. Uniformly quantise again using fewer bits.
4. Transmit.

You probably know that μ-law and A-law are companding algorithms that serve to make the gaps between the lower quantised values smaller, and those between the larger ones bigger.

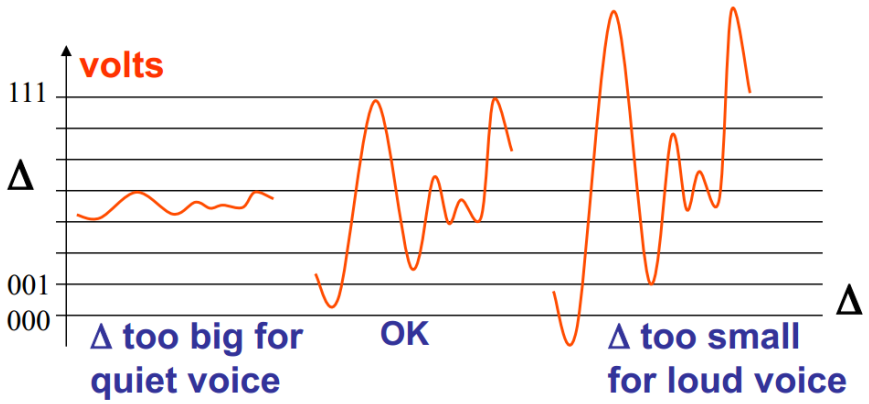


Figure 6: This waveform shows how a a bad choice for Δ can adversely affect the quality of the sound, when it is undergoing uniform quantisation.

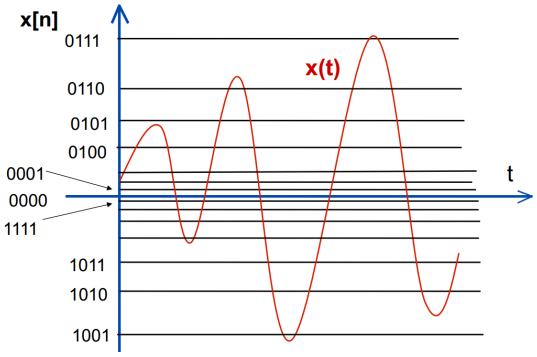


Figure 7: An example of non-uniform quantisation.

5. Reverse the process of steps three and two with an 'expander'.

A 'compander' will increase the value of small samples, and decrease the value of large samples, and an 'expander' just does the reverse. Even though we only apply two uniform quantisations, the compander and expander mean the overall effect is one of non-uniform quantisation.

Companding is just a coding technique similar to compression. It doesn't increase the quality of the sound or anything (though it might do if the alternative was to use the same number of bits with uniform quantisation.)

4.2.1 Quantisation error

Quantisation Error is produced when we quantise a wave, and is random in the range of $\pm\frac{\Delta}{2}$. Since the error is random, the error is heard as white noise, and is spread evenly across all frequencies. The Mean Square Value (MSV) of the noise, is $\frac{\Delta^2}{12}$.

We can calculate the Signal to Quantisation Noise Ratio (SQNR), which is how loud the signal is in comparison to the noise (hence it is given in decibels - see Section 2.3). To do this, we use the formula:

$$\text{SQNR} = 10\log_{10}\left(\frac{\text{MSV of signal power}}{\text{MSV of noise}}\right)$$

We can use this formula to calculate the dB/bit for sending telephone data (speech, text etc):

$$\text{SQNR} = 10\log_{10}\left(\frac{A^2/2}{\Delta^2/12}\right)$$

The maximum amplitude (A) of a sine wave that has been quantised with a step size of Δ is $\frac{2^{bits}}{2}\Delta$

$$= 10\log_{10}\left(\frac{\left(\frac{2^{bits}}{2}\Delta\right)^2/2}{\Delta^2/12}\right)$$
$$= 10\log_{10}\left(\frac{\left(2^{2\times bits-2}\Delta^2\right)/2}{\Delta^2/12}\right)$$
$$= 10\log_{10}\left(\frac{\left(\frac{2^{2\times bits}}{4}\Delta^2\right)/2}{\Delta^2/12}\right)$$
$$= 10\log_{10}\left(\frac{\frac{2^{2\times bits}}{8}\Delta^2}{\Delta^2/12}\right)$$
$$= 10\log_{10}\left(\frac{\frac{2^{2\times bits}}{8}}{1/12}\right)$$
$$= 10\log_{10}(1.5\times 2^{2\times bits})$$
$$= 10\log_{10}(1.5) + 10\log_{10}(2^{2\times bits})$$
$$\approx 1.8 + 10\log_{10}(2^{2\times bits})$$
$$\approx 1.8 + (6\times bits)$$

This only strictly applies for uniformly quantised sine waves, but also holds fairly well for speech and music.

If an 8 bit uniform quantisation scheme is designed for loud talkers (so they will use all eight bits), then it will have a SQNR of $(6\times 8)+1.8 = 49.8$dB. If another talker is 30dB quieter, then the samples will be encoded using only 3 bits:

$$\begin{aligned} 6\times bits - 1.8 &= 49.8 - 30 \\ 6\times bits &= 48 - 30 \\ 6\times bits &= 18 \\ bits &= 3 \end{aligned}$$

Since the quantisation noise for three bits is much lower (19.8dB) for the quietly talking person, they will probably be able to hear the noise over the phone.

For the loudest CD quality (16 bit) music, the maximum SQNR value is 97.8dB, whereas for the most quiet sounds, it is -22.8dB (which means the noise is louder than the actual sound). Obviously this isn't ideal, so we need to apply DRC (Dynamic Range Compression) improve the balance.

Remember, since the SQNR is calculated by $\frac{\text{MSV of signal power}}{\text{MSV of noise}}$ it has an inversely proportional relationship with the amount of noise. I.e. will go up as the noise decreases.

Dynamic Range Compression is where you reduce the volume of loud sounds, and/or increase the volume of quiet sounds which has the effect of reducing the dynamic range of the audio signal.

4.3 Differential encoding

A sample of audio isn't just random data, and since the data represents a wave, consecutive values are likely to be relatively close together. If that is the case, then maybe we could encode the data as the difference between one sample and the next. If the differences are easier to transmit than the raw data, then we could save resources such as bandwidth. Such an encoder is shown in Figure 8.

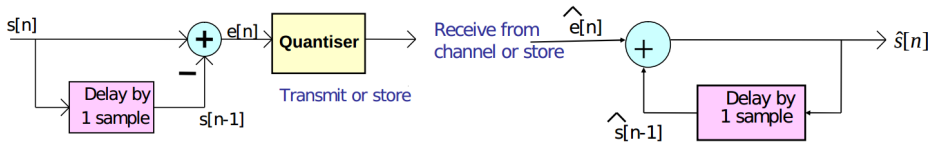


Figure 8: An example of how a differential encoder might both encode and decode audio samples.

PCM stands for Pulse Code Modulation, and is basically the same as what we mean by quantisation.

This is known as Adaptive Differential PCM, and can be made to work at bitrates of $16-32$kbit/s. However, for use in mobile telephones, we need a bit rate that is at least four times lower than this...

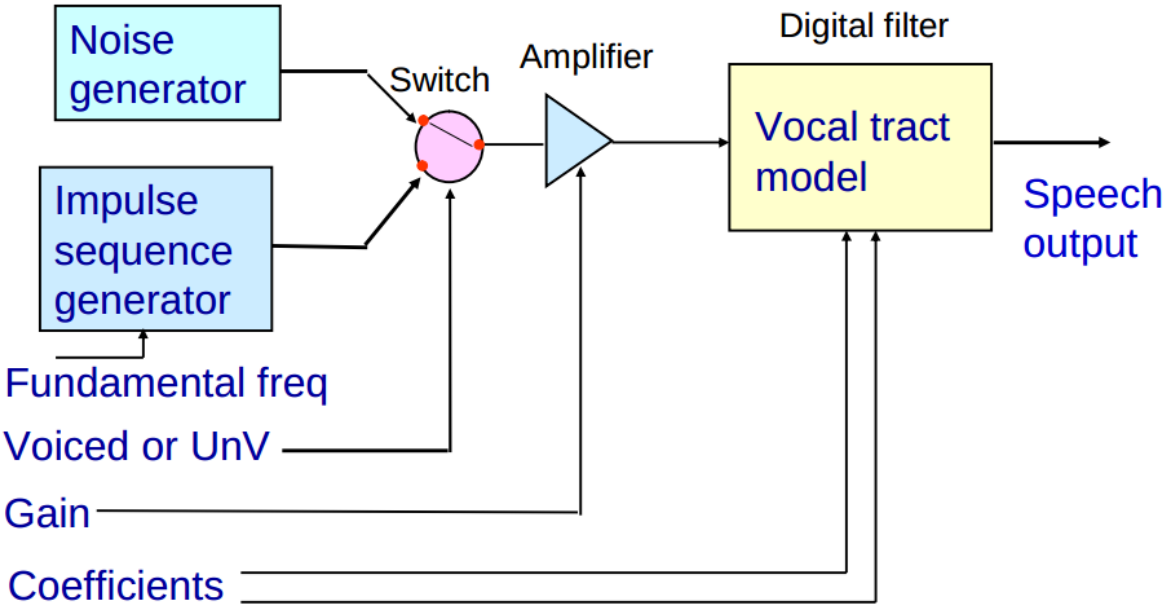


Figure 9: An implementation of a Linear Predictive Speech Decoder

4.4 Linear Predictive Speech Coding

Voiced speech is normal speech, unvoiced speech is whispering. If normal speech is sampled, then some of it will be classed as unvoiced, since only vowel sounds actually require the use of vocal chords (try it!).

LPC is a type of parametric encoder. Working on the principle that voiced speech is created by the vocal chords opening and closing at frequencies that change over time, the coder can send the fundamental frequency of the noise as well as parameters (called coefficients here) modelling the shape of the mouth over the network to recreate the speech at the receiver.

The sender derives the parameters at a rate of 50hertz, which include how loud the speech is, the coefficients modelling the vocal tract, and the fundamental frequency of the sound. Figure 9 shows how these parameters are used to reconstruct the speech at the receiver.

4.4.1 LPC-10

LPC-10 was a coder once used by the military that had a bit rate of 2.4kbit/s. Each 20ms frame has the following components:

37 bits	Ten filter coefficients
1 bit	Voiced/unvoiced decision
8 bits	Gain (the amplitude)
8 bits	Fundamental frequency

MIPS is Millions of Instructions Per Second

This made for a total of 56bit per frame. Although it's easy to understand and implement, and requires little processing power (according to Wikipedia[1], it requires 20MIPS and 2kB of RAM), it sounds horrid.

When the speech is unvoiced, then a consonant is being transmitted; vocal chords are not active when this happens and so a random signal can be used in place of the fundamental frequency.

4.4.2 Codebook Excited LPC

Codebook Excited LPC (CELP) is a way of further reducing the bandwidth used by LPC. The principle is that there are a number of pre-configured parameters that the receiver and sender have stored in a 'codebook', and the sender just sends the index of the one that is most similar to the sound that it is trying to produce.

[1] http://en.wikipedia.org/wiki/FS-1015

It finds the most similar sample using 'analysis by synthesis', which involves trying all the entries in the codebook one by one, and sending whichever is the closest to the one we want.

CELP is used in commercial mobile telephones, and is utilised by the **Adaptive MultiRate** (AMR) coder for rates at 12kbit/s or lower.

4.5 Comfort noise

When a two way conversation is going on, and neither party are speaking, nothing will be played through the speakers to either party (since we don't transmit quiet sounds since that wastes power). To mitigate this, the phones will generate and insert 'comfort noise' which is psudorandom background noise that sounds like whatever is going on at the other end of the phoneline. Each phone will determine if its owner is silent, and transmit the characteristics of the background noise using very few bits if they are so that the other phone can recreate a similar background noise for the other person.

4.6 Error correction, detection & prevention

Mobile systems are affected by errors in the transmission of data, often in the form of noise affecting radio reception. There are many ways of avoiding/mitigating errors, the most important ones looked at in this course are; Forward Error Correction (FEC) and error detection and retransmission (Automatic ReQuest - ARQ).

4.6.1 Forward Error Correction

We can build redundancy into the transmission by appending check bits or some other form of coding that will bloat the size of the transmission, but result in less errors.

We can use block coding or convolutional coding to implement this. **Block coding** is when you process the data in blocks (well duh), which requires you to encode the whole block at the transmitter and decode it again at the receiver. You can't use a block unless it's been fully decoded. **Convolutional coding** on the other hand can yield usable bits soon as just a few of the transmitted bits arrive at the receiver. Convolutional coders treat data as a stream, whereas block coders deal with data in chunks.

Here are some examples of the above:

Repeating bits:
: We could simply send bits a few times, instead of just once and take the mode of them at the receiver. If we sent three bits for every bit, then if there was one bit error, then it would be fine since we could take the majority of the received bits, and the one error bit would be ignored.

  The number of repeats must be an odd number, otherwise we could end up with a half and half split.

Parity
: As a simple (and not very effective) way of detecting bit errors, we could append a parity bit to the end of blocks. The parity bit would be `0` if the block had an even number of ones, and `1` if it had an odd number of ones. If we have an n bit number, we can calculate the parity using $n-1$ `XOR` operations; $b_0 \oplus b_1 \oplus \cdots \oplus b_n$.

  When the receiver gets the bits, the parity of all of them (including the parity bit) must be computed. If the parity is `1`, then some bit error definitely occurred, whereas if it is `0`, then the data *may be* correct (but there could be two or more errors as well).

  Parity checking is a block code, since you need to wait for the whole block to arrive before checking the parity.

Hamming distance
: The Hamming distance between two binary numbers is the number of bits that are different. This is obtained by `XOR`ing the two numbers together and counting the number of ones. This is useful if we want to know how many bit errors it takes to convert one number to another.

You have a set list of code words that you can use, each with a minimum hamming distance from it to any other. Then, you can detect and even errors by comparing your received code to the allowed ones.

There is an implementation on page 32.

Hamming codes are block codes, since you need the whole block to calculate the distance between the received block and the ones in the codebook. It is common practice to introduce check bits onto code words to make the distance larger.

The notation for Hamming Codes is $(m + r, m)$, where m is the number of message bits, and r is the number of check bits. The r bits are appended to the end of the code words, and are derived from the value of the m bits. For example, with a $(7, 4)$ hamming code, we could make:

- $r_0 = m_0 \oplus m_1 \oplus m_2$
- $r_1 = m_0 \oplus m_1 \oplus m_3$
- $r_2 = m_0 \oplus m_2 \oplus m_3$

This would give a *syndrome table* (i.e. correction table) of:

r_0	r_1	r_2	Correction to
0	0	0	None
0	0	1	r_2
0	1	0	r_1
0	1	1	m_1
1	0	0	r_0
1	0	1	m_2
1	1	0	m_3
1	1	1	m_0

Interleaving:
Since bit errors in radio links often occur in bursts (e.g. caused by a car turning on), we could transform one dimensional blocks of data into two dimensional matrices, and transfer them column by column. This way, if there is a 'bursty' bit error, then it will affect multiple rows, but the bit error correction such as the repeating method, or hamming codes could detect the error since only one or two bits may be wrong, not all of them.

4.6.2 CRC checks

A Cyclic Redundancy Check is a block code for detecting bit errors. If we find the value of the number we're transmitting in decimal and divide it by a number such as seven, we can append the remainder onto the message. Then at the receiver, we can divide the received bits again and if we get a different remainder, then we know that a bit error has occurred.

If the bit errors happened to add or subtract the number that you were dividing by, then the remainder would be the same and you would not be able to detect the errors. If we make the number large, then this is unlikely.

Real CRC checks use *Galois Field Arithmetic*[2] to do the maths, since binary numbers can be expressed as polynomials:

Summing:
To calculate the sum of two binary numbers, we can just XOR their bits. Subtracting is the same:

$$1001 \oplus 0111 = 1110$$

Long division:
This isn't explained particularly well by the lecture slides (mainly because multiplication isn't explained, which is helpful to understand for division) and I haven't got time to explain it before the exam, sorry! Ask Barry :)

There are three CRC generators used in practice:

[2] `http://en.wikipedia.org/wiki/Finite_field_arithmetic`

- CRC-8-ATM: $x^8 + x^2 + x + 1$
- CRC-16-IBM: $x^{16} + x^{15} + x^2 + 1$
- CRC-32-IEEE: **Quite long...**

A generator of order r can detect all error bursts of length $\leq r$.

4.6.3 Convolutional Coders

If we are processing streams of data, then we could have a 'rolling parity check'. When we encode, the previous n bits are `XOR`'d together to produce another stream of bits. We can then interleave the new parity stream with the normal stream.

At the decoder, the bits are valid if each of the parity bits (i.e. not the normal ones) is the `XOR` of the previous three normal bits. If the bit sequence is invalid, then we can select the valid sequence with the minimum Hamming distance to the received sequence and therefore have error correction.

With sequences of around 8 bits, that is fine, since there would be 256 valid sequences of 16 bits long each, however, for longer sequences (e.g. 1024 bits), then this would not be feasible.

Convolutional coders that encode bit streams are called *rolling parity* coders.

A soft decision decoder is where if we're unsure as to what the value of a bit is, then we could use previously known probabilities to determine what we could pick.

4.6.4 Miscellaneous FEC stuff

Well thought out use of FEC techniques in mobile systems increases the energy efficiency and effectiveness of the system. Transmitting at higher power is one way of reducing errors in the system, but 'loud' signals also cause lots of interference over a wider range and, of course, uses lots of battery. FEC lets us reduce the transmission power, yet overcome the bit errors that come as a result, which allows us to re-use different frequencies and reduce power consumption.

We could increase the bit rate by not encoding in binary, but using any 2^n number of states. If we could send the values $\{0, 1, 2, 3\}$, then we could send two bits per pulse, effectively doubling our bandwidth. This is limited by the noise in the channel.

The **Shannon-Hartley Law** says that the channel capacity C is equal to:

$$C = B \log_2(1 + \frac{S}{N})$$

Where B is the bandwidth in Hz, and $\frac{S}{N}$ is the signal to noise power ratio. C is given in bits per second.

4.7 Image Coding

Images are really big. A 10"x8" colour photo at 300dpi has $10 \times 300 \times 8 \times 300 = 7.2$Mpixels. Since each pixel requires three bytes; one for each of Red, Green and Blue (RGB), we need 21.6MB to store the image.

If images are so big, think how big movies must be! TV quality video is at 25Hz, where each frame is 640×480. This means we need $640 \times 480 \times 3 \times 25 = 23.04$MB/s for a (low quality) movie. If a movie lasts two hours, this is about 160GB.

Thankfully, we can exploit deficiencies in human vision to reduce these numbers a lot. For example, we can either represent a pixel by using a separate value for red, green and blue, or we can use luminance (monochrome) and two chrominance (colour) components.

Our eyes are less sensitive to chrominance than luminance, and are also less sensitive to rapidly changing fine-detailed aspects of movies. We will exploit this to save bits later.

4.7.1 Frequency domain processing of images

Just like sound, we can convert images into the frequency domain and process them there. Using this, we can efficiently encode only the features that will be perceived by the eye. We need to use a two dimensional Discrete Cosine Transform instead of just a one dimensional DCT like we do with sound (for obvious reasons).

It's best to process images using luminance and chrominance instead of RGB. To convert between RGB and YIQ (Y is for the luminance, I is for in-phase and Q is for quadrature), you do:

The numbers might differ slightly, but the idea is the same.

$$
\begin{aligned}
Y &= 0.30R + 0.59G + 0.11B \\
I &= 0.60R - 0.28G - 0.32B \\
Q &= 0.21R - 0.52G + 0.31B
\end{aligned}
$$

When we've converted an image into YIQ and ran it through DCT, we can plot a graph of the values to get something like in Figure 10. It is easy to see that there is a lot of energy at the low frequencies and not much at the high ones. If we zero the low values, then we get something from the right of Figure 10.

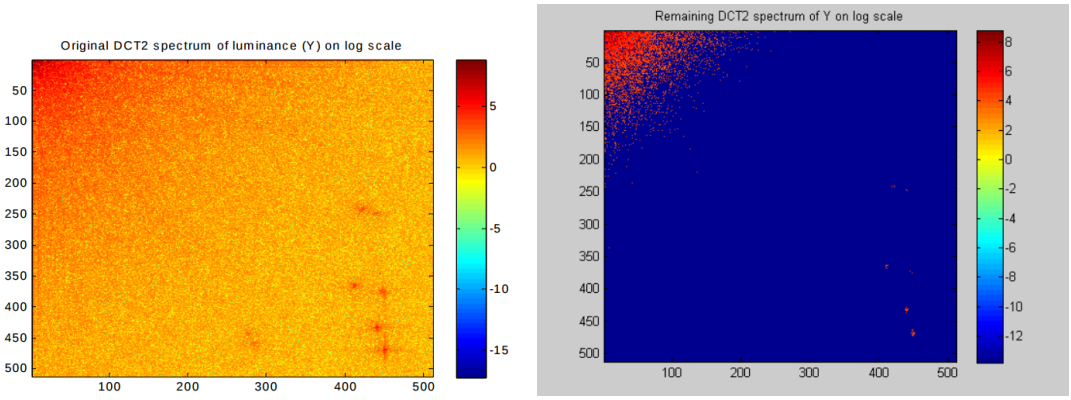


Figure 10: DCT Luminance Spectrum (left) and the DCT Luminance Spectrum after the small amplitudes have been set to zero (right).

For this data set, the number of non-zero coefficients for Y was reduced from 262000 to 8692, meaning that only 3% remained! If its similar for I and Q, then we've got a very large compression ratio. Unfortunately, the quality isn't satisfactory when you do this (though it's not *bad*)

4.7.2 The JPEG standard

JPEG divides images into 8×8 coloured tiles. For each tile, the values are converted into YIQ from RGB and the chrominance is reduced to a 4×4 resolution by averaging the values (since our eyes are less sensitive to chrominance).

Each tile is then run through a 2D DCT that is then quantised into an integers, with different frequencies being quantised by different amounts according the eye's sensitivity to them, as shown in Figure 11.

Quantization is one of the things that makes JPEG (and therefore MPEG) a lossy compression algorithm. The loss of quality is small, but not zero. You can however change the quality/compression trade off by adjusting the quantization matrix.

The algorithm then takes the average value for the tile and encodes it as a difference from tile to tile. This usually changes very slowly, but is noticeable in the image. We can then encode these coefficients in a zig zag pattern and use run length encoding, as in Figure 12

We can then apply Huffman coding to the remaining bits (after they've been run length encoded) to save more space.

4.7.3 Huffman Coding

A Huffman code is a variable length, self terminating code. Basically, if you know the probability of a specific word coming up in some data is high, then you can agree to replace that word with some

DCT Coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 11: JPEG images are quantised into integers of differing accuracies (i.e. bits) for different frequencies

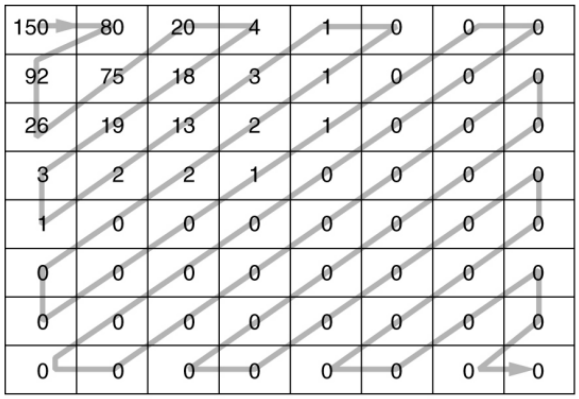


Figure 12: JPEG encodes coefficients in a zip zag pattern to optimise for run length coding

smaller word and you'll save bits. You do this by creating a *Huffman Tree*. Given words and their probabilities, a Huffman tree generates a binary tree, where a 1 takes you right, and a 0 takes you left. When you reach a leaf, it will contain the value of the word, and the series of 1's and 0's you used to get there is the shorter replacement.

To generate the tree, the algorithm ranks the words in descending order of their probability, and links the two with the lowest. It keeps doing this (ordering the changes when necessary) until all the words are in the tree. Figure 13 shows this.

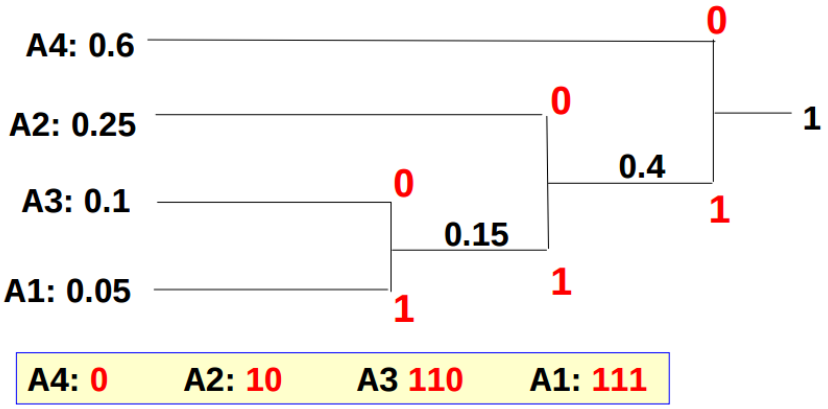


Figure 13: A generated Huffman tree.

The trees generated by a Huffman coder are sometimes perfectly balanced binary trees, in which case the code length is not variable and all the codewords have the same length.

Huffman coding is highly efficient and lossless, but is very sensitive to bit errors because of its variable length and the fact that its words are self terminating. If there is a bit error in one code word, then it could result in us terminating it at the wrong moment, and starting the next code word at the wrong place, which will lead to the whole message becoming corrupted!

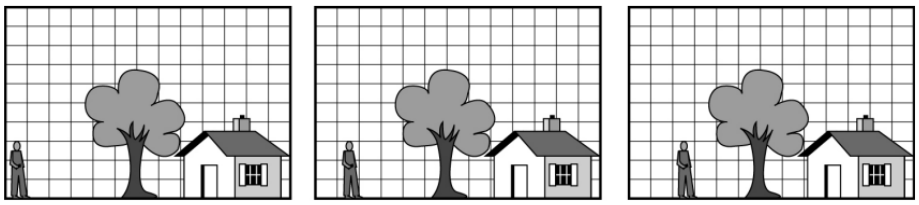


Figure 14: You could save bits encoding these three frames by encoding that the man is moving, rather than encoding the whole frame again for the second and third frames.

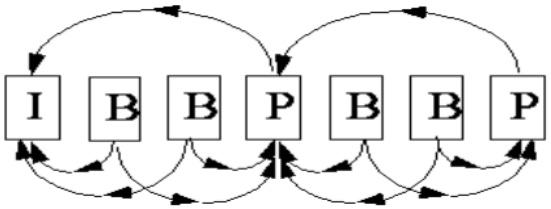


Figure 15: How I,P and B frames work together to save data!

4.7.4 Encoding Movies

If we were to encode each frame in a movie as a separate JPEG, then it wouldn't be efficient at all since most frames are very similar to those before and after them. Maybe we could just send the differences between consecutive frames, but this wouldn't work for when a scene is panning across a landscape or zooming in for example.

A better approach is to use 'motion compensation' which is where you find similarities between parts of images in successive frames and send motion information when the similarity is strong. The differences are then encoded as normal using JPEG. An ideal candidate for this would be something like in Figure 14.

MPEG videos have three types of frames:

I-frame:
This is an image frame encoded as a JPEG

P-frame:
These frames encode the positions of moving blocks from previous P and I frames and the remaining differences between the frames.

B-frame:
The positions of moving blocks from previous **and** next P and I frames and the remaining differences are encoded by B-frames.

Figure 15 shows how these fit together.

MPEG also uses a clock to synchronize the audio with the video. A multiplexer encodes the audio and video together into one MPEG output stream.

5 Mobile Networks

Since their inception, the desired use of cellular networks has changed from being solely about transmitting speech, to transmitting textual data, and now arbitrary packets. This has required them to evolve from 'circuit switched networks', where (at a fundamental level) a physical connection must be made between wires to facilitate data transfer, to 'packed switched networks'.

This evolution is really interesting, since circuit switching originated from landline phones, but packet switching was from digital networking such as Ethernet. As the technology has progressed, cellular data has moved towards the Internet's way of doing things.

However, mobile phones have moved on from being single feature devices, merely letting you call other users. Now they are *smart*phones, and this requires them to use a plethora of different technologies to be able to fulfil user's demand; GPS, bluetooth, radio etc to name a few.

5.1 Cellular networks

One reason why mobile phones can work so well is that they take advantage of *spatial multiplexing*. This divides an area into small areas called cells, each having a frequency band. The cells are different sizes based on their expected usage rates (cells will be more dense in areas with more demand). Frequency bands are re-used when a cell with the same frequency band is far enough away, and because of this, mobile phones must take care not to transmit their signals too loud and pollute the airwaves for nearby cells of the same frequency.

Cells are typically $0.1 - 35$km in diameter. Obviously small cells are better since they will handle more users per unit area (assuming cell size is independent of user capacity), but users will need to transmit at a lower power (so the neighbouring cells using the same frequency aren't affected). This is actually a good thing sometimes, since it uses less battery, and we can use FEC to mask bit errors.

Breaking the spectrum up into frequencies, and mapping that onto physical cells is a nice concept - and it works very well in reality, but it does have some drawbacks; for example, what happens when a user moves into a different cell while on the phone? Thankfully, the system supports transparent and seamless handovers as the user moves from cell to cell.

5.1.1 The evolution of cellular technologies

Cellular networks have gone through four iterations to date, with a fifth one coming in the future:

0G:
The zero'th generation of phones were merely radio telephones. They didn't use cells at all.

1G (1983):
This is when cellular mobile was created, all signals were analogue.

2G (1991):
Data is introduced, though it is very slow. In 1998, GPRS was introduced, which brought speeds from $56 - 114$kbit/s and in 2003, EDGE was introduced which has speeds of up to 384kbit/s. These are called 2.5 and 2.75G respectively.

3G (2001):
This introduced better speech and faster data. 3G has multiple revisions; 3.5G (2007) is HSPDA with speeds of $1.8 - 7.2$Mbit/s download and 384kbit/s upload, 3.75G (2010) is HSPA+ which has speeds of 56Mbit/s download and 22Mbit/s upload, and finally 3.9G which is still being created and launched, and is related to technologies such as WiMax and LTE.

4G (2011):
The specification for 4G is 1Gbit/s download and 100Mbit/s upload. However, the technology doesn't meet that yet (even now!). Henceforth, companies marked 3.9G technologies as 4G since they 'aspire' to meet the target speeds.

5G:
Due to be launched around 2021, though we've still not met the 4G speeds yet...

This isn't the whole story though. It wasn't just the marketing names and the speeds that changed, they also used completely different multiplexing techniques to let multiple users share the spectrum:

All of these use spatial multiplexing with cells except 0G.

0/1G:
Frequency Division Multiplexed Access (FDMA) was used, which is where each transmitter is given a different carrier frequency.

2G (in Europe - GSM):
Time Division Multiplexed Access (TDMA) was used so that each transmitter is given a regular time slot to send data in.

2G (America only)/**3G**:
Code Division Multiplexed Access (CDMA) - each transmitter uses the same frequency band, but different codes are used to send data.

4G:
Orthogonal Frequency Division Multiplexed Access (OFDMA) is used by 4G, which utilises

multi-input/multi-output (MIMO) antennae to transmit on multiple frequencies at once. Data is sent in packets.

5.1.2 2G in detail

2G implements the features shown in the next list. They are linked together using buffers shown in Figure 17.

TDMA:
Each TDM frame is 4.615ms long, and each mobile can send 114bit of data in every frame. Since there are 217 frames/s, the bit rate is $114 \times 217 = 24.7$kbit/s. If FEC is used, then the data rate drops to 13kbit/s.

Remember, not all of the capacity of the network is there for the user since some is left for synchronisation and signalling.

FDMA:
There are two 25MHz channels split into 0.2MHz bands. One channel is for base station to mobile and the other is for mobile to base station.

Speech Transmission:
The device captures the analogue voice, runs a low pass filter over it. The analogue wave is then sampled, quantised and packetised into blocks of around 20ms. Linear Predictive Coding is then used to reduce the bit rate to around 260 bit/block (the equivalent of around 1.5 bits/sample), which gives a bit rate of around 13kbit/s.

Remember, a low pass filter keeps low sounds and attenuates (removes) higher frequencies greater than half the sampling frequency.

When the TDMA time slot happens, 114 bits are read from the buffer, modulated into a sinusoidal carrier wave, and transmitted via the antenna.

Receiving Speech:
First of all, the signal is demodulated and 114 bits are extracted into the receiver buffer. Then, at 20ms intervals, 490 bits are extracted from the same buffer, are decrypted and the interleaving is removed. Bit error correction is applied and FEC is removed to leave 260 bits, where we can then apply the LPC decoder to get 160 samples that are then converted back to analogue, amplified and sent to the speaker.

Binary Frequency Shift Keying (B-FSK):
Binary FSK is used to transmit data in a sinusoidal wave. A low frequency wave is sent for 1 and a high frequency one is for 0. It is simple, not very efficient, but has a constant amplitude. This is shown in Figure 16.

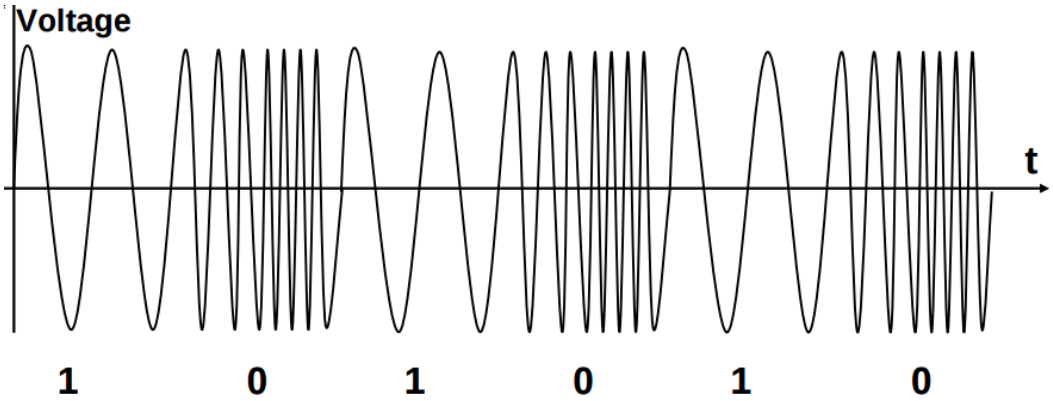


Figure 16: How the data `101010` might be encoded using BFSK. If you try and sing this wave, then you sound like Tarzan.

5.1.3 3G in detail

3G uses CDMA instead of TDMA to provide support for multiple users, but in many ways, it is similar to 2G. The main difference is the use of **CDMA**:

Code Division Multiplexed Access is used to let multiple carriers transmit on the same frequency. Using this technique, each bit is turned into a sequence of chips that are transmitted instead, maybe:

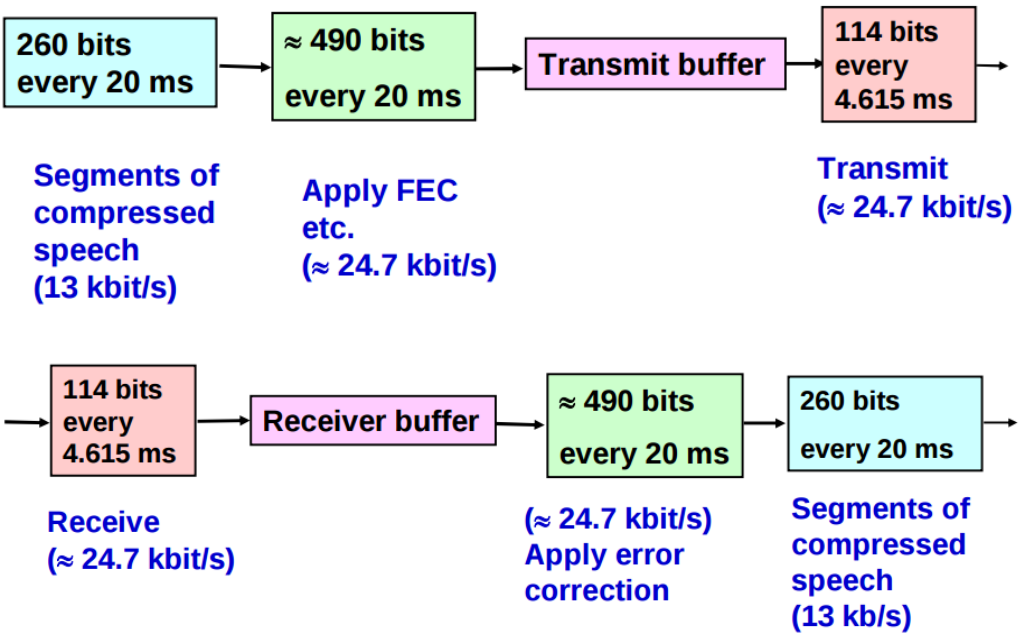


Figure 17: The sending and receiving buffers in a typical 2G implementation.

```
1 = 1 0 1 0 1 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1
0 = 0 1 0 1 0 0 1 1 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0
```

In this manner, every bit becomes a psudo-random sequence of chips, which lets the receiver distinguish different carriers from each other using a cross correlation process. This requires a much larger bit rate than TDMA, but carriers can transmit at any time though. Another advantage of CDMA is that it has a soft capacity limit; the quality merely degrades when more people use it (since the background noise increases as other carriers transmit seemingly random signals).

Lets say that User A has a code of:

```
1 = 1 -1 1 -1 1 1 -1 -1 -1 1
0 = -1 1 -1 1 -1 -1 1 1 1 -1
```

When we receive a signal, we should multiply it by the code for 1 (`1 -1 1 -1 1 1 -1 -1 -1 1`) and sum the bits, and we get $+10$ for 1 and -10 for 0.

User B should use a different code:

```
1 = 1 1 1 1 1 -1 -1 1 -1 -1
0 = -1 -1 -1 -1 -1 1 1 -1 1 1
```

If we multiply a code received by B's code for 1, then we should get $+10$ or -10 again if B sent it. If we multiplied and added using A's code, then we would get 0.

If A transmitted a 1 and B transmitted a 0, then we'd get:

```
0 -2 0 -2 0 2 0 -2 0 2
```

If we multiply and add using A's code, we get $+10$, and if we do it with B's code, we get -10.

Here are some reasons why 3G's CDMA is better than 2G's TDMA:

- TDMA requires you to strictly synchronize transmission times of all users to ensure that they're in the correct time slot. *Guard times* are used between slots to ensure there is no overlap, but this decreases spectral efficiency.
- CDMA allows for a flexible allocation of resources; there's only a soft limit on the amount of users you can have.
- Channels that aren't being used on 2G are wasted (they're just empty slots aren't they). Less capacity on CDMA just means less interference for users and fewer bit errors.

Really, the only place where CDMA might loose out to TDMA is power control. This is difficult to achieve with CDMA, especially since signals near to a receiver might drown out a signal far away that you're trying to listen to. This is called the *near-far* problem.

5.1.4 The details of 4G

As previously mentioned, 4G uses Orthogonal Frequency Division Multiplexed Access, but it also implements MIMO and packet switching:

OFDMA:
: This is when the sender transmits on many sinusoidal carrier waves at the same time (for each receiver). Data is dispersed among them, so that if some waves are not received properly, then it can be obtained from the others.

Packet switching:
: Only at 4G, has mobile telephony moved away from connecting wires to facilitate communication. Now, all speech and data is sent in packets and conveyed using the Internet Protocol (IP).

Multiple Input, Multiple Output:
: The capacity of the channel is doubled by having two transmit and two receive antennas. This is also used by **WiFi**.

  The idea of MIMO is that if more than one transmitter and receiver is used, then the capacity of the wireless link will be increased. The transmitter will modify the signal of each so that the sender can distinguish which sent what signal, and besides, each signal will reach the transmitter via a slightly different path causing a different frequency loss and delay.

5.1.5 5G and the future!

With 4G still not ready (it doesn't meet the speeds that were expected for 2010 as of 2015), 5G development is more in the ideas stage rather than the implementation stage, however, the ideas are still really cool.

It could use BDMA (we'll come to that in a second), smart radio to use different parts of the spectrum based on current usage, IPv6 (as opposed to IPv4 which is the currently implemented, but outdated standard), the option of device to device communications (bypassing the base station entirely) and more cool stuff.

Smart radio is also known as *cognitive radio* and is a form of *dynamic spectrum management*.

Beam Division Multiple Access is when you use multiple (e.g. 7) antennas spaced at equal intervals to make amplitude of the signal higher in some places than at others. See Figure 18 for a pictorial explanation (you might have to think back to A-level physics to understand why this happens).

The transmitter can change the amplitude and phase of the transmitted signal from each of the antennae to form the signal to be strongest at the receiver. Other receivers can be nulled out, or at least sent a reduced signal. This works both ways; a receiver that has seven antennae can target a specific transmitter. It's seen by many as a better concept than cellular multiplexing, more like having actual wires.

Group Cooperative Relaying is a technique where a separate carrier between the transmitter and receiver can boost the signal by repeating it and acting as a relay. This can be performed in two ways:

- *Amplify and forward* sends the raw signal on again, but also amplifies the noise.
- *Decode and forward* decodes the transmission, re-codes it again and sends it on. This has the benefit of reducing noise since the signal is forwarded on with the noise removed.

5G could also be really **energy efficient**; it is proposed that as load on the network decreases, some cells could be turned off, and the coverage area of other cells could be allowed to increase. The network can adapt to the current usage pattern in real time. Care must be taken though; since transmitting to base stations further away will require more battery power from the user's phone which is obviously a bad thing.

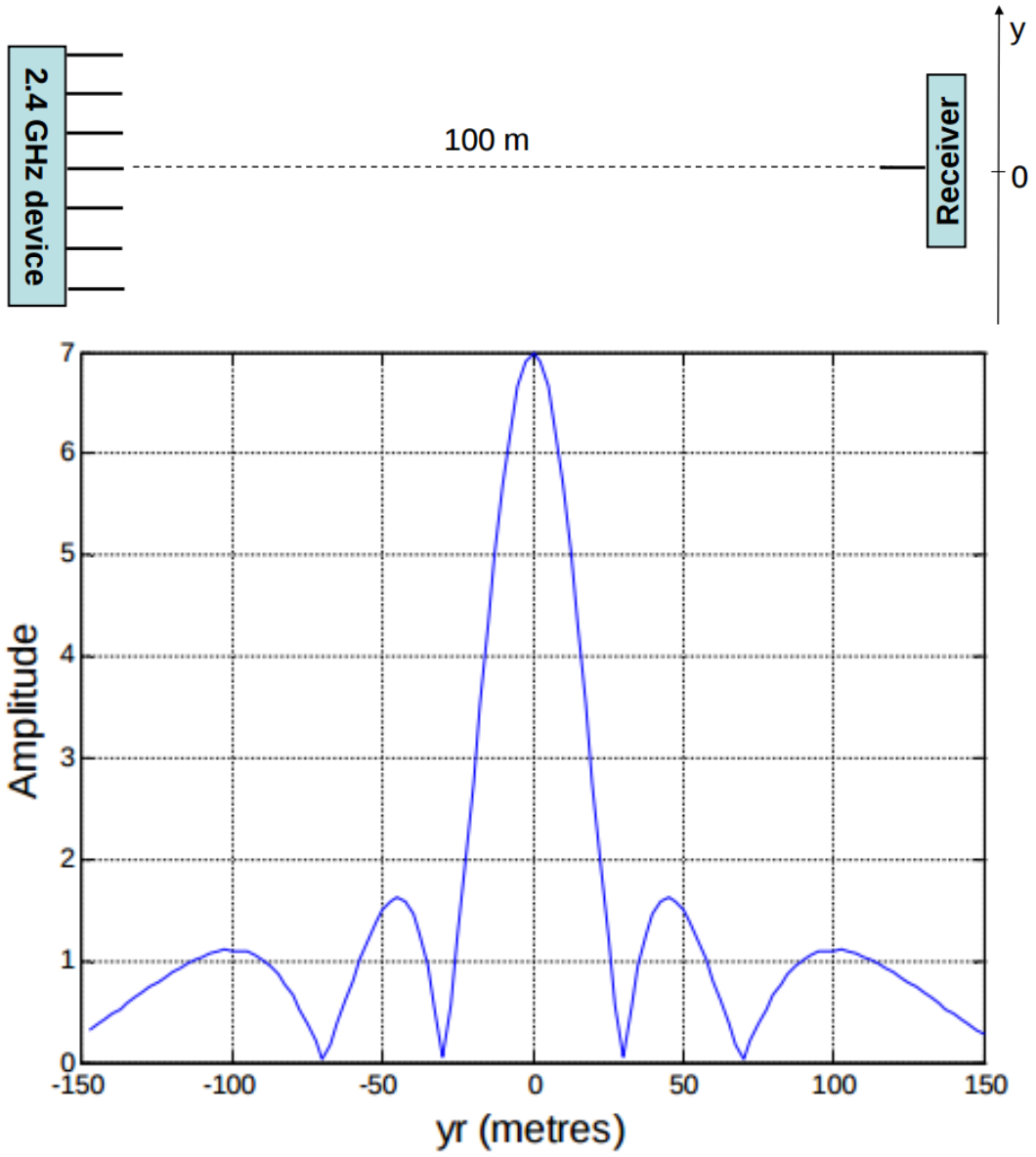


Figure 18: Here, you can see that as we adjust y, the intensity changes.

5.2 GPS

GPS stands for Global Positioning System, and it works by having a receiver calculate its position from timing signals sent by GPS satellites that are circling about 20000km above earth. They send messages that give the exact time and the position of the satellite, which allows the receiver to determine its distance from the satellite by the offset between the transmission time and received time. If at least four satellites can be 'heard' by the receiver, then its position can be accurately found.

5.3 Multimedia communications

We need to efficiently, reliably and sometimes in real-time, send large amounts of data. We can use compression to reduce the volume of data with acceptable quality/volume trade-offs, and bit error correction to allow us to reduce the transmit power to make this easier. In the end, it's a series of trade-offs between power, bandwidth, latency, quality and more factors.

To avoid disappointing the user with a poor quality link, smartphones will often download a whole file and then use it. It's not real time, but the user will have uninterrupted access to the file when its on the device. However, the file could take a (long) while to download, and it could also take up valuable space on the device.

One example of this may be if the user wanted to download a video for a train journey where their data connection might be patchy for the duration of the journey (maybe it's a long tube journey in London).

In theory, the bandwidth of the data connection is enough to stream the content in real time especially with 3/4G.

Usually though, two things are true about the situation; firstly, the bandwidth of the data connection is higher than or similar that of the file being downloaded (e.g. a mp3 file or a movie), and secondly, the data connection may not always be available for the duration that its needed (i.e. when you're listening to a song).

We can take advantage of a *buffer* to smooth out the jitter of the data connection. We can start the media player as soon as we have a sufficient reservoir of data. This analogy extends nicely to when we want to get more data, and when we don't need it; when the reservoir of data is getting low (maybe we have a minute of data left in the buffer), then we should request more from the server, but when the buffer is nearly full, we should stop asking for more data, otherwise we might have to 'bin' some of it since we wouldn't have room on the device. See Figure 19 for a depiction.

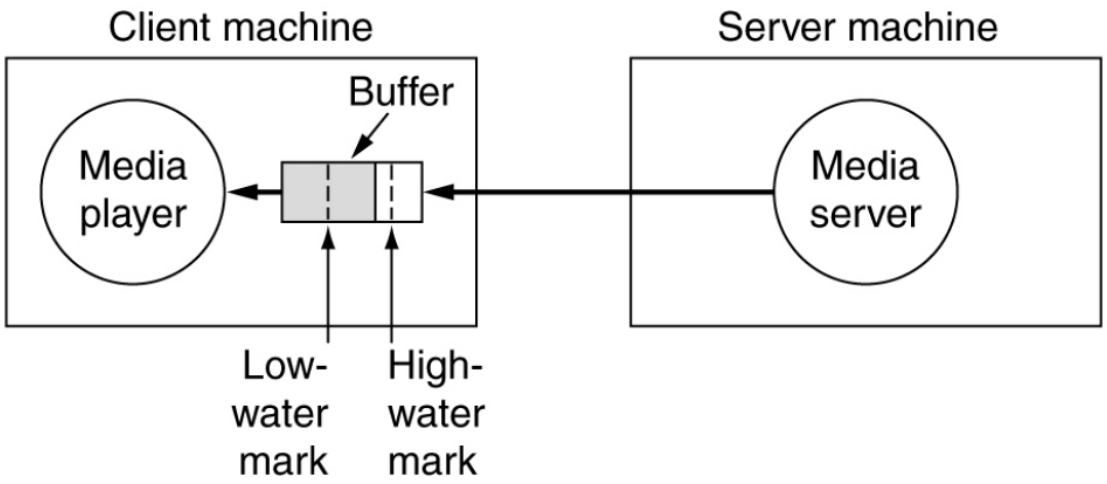


Figure 19: We start downloading at the low water mark and stop when we reach the high water mark.

5.3.1 Streaming protocols

Protocols such as TCP are a bit overkill for streaming. We generally (well, not as much) don't need to resend lost packets, or protect too much against bit errors, since both cases are less critical for streamed data.

To protect against lost packets, we can put alternate samples in neighbouring packets. If one packet is lost, then we can interpolate the values in the received packet to smooth over the loss and get an approximation for the lost packet, as shown in Figure 20.

5.3.2 VoIP Telephony

Since VoIP is interactive, we treat it slightly different to streaming data. First of all, we need to make sure the round trip time for packets is less than around 0.5s, since otherwise it will make participants talk over each other.

Because of this requirement, we can't use TCP since any retransmissions will probably be too late, so instead we use RTP. This is a "fire and forget" protocol, since if it is lost, then we don't mind. We can tolerate bit errors and packet loss by interpolation.

Since the LTE standard (aspiring to 4G) only supports packet switching, we have to use VoIP to talk to people!

Though phones using LTE currently move down to 2/3G to talk

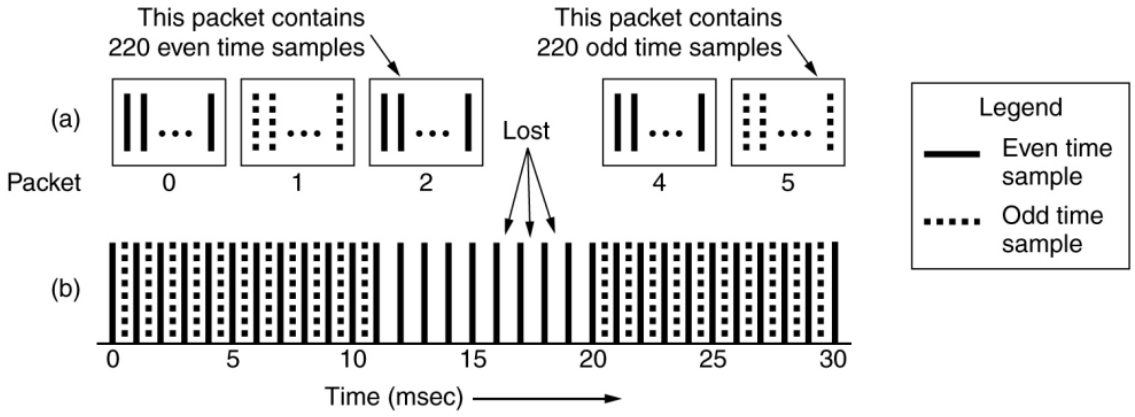


Figure 20: Packet 3 was lost, but we can figure out an approximation for what was in there by interpolating the data from packet 2.

5.3.3 Optimising power

As you've probably now worked out, there are a lot of trade offs when building mobile systems. For example:

- Better quality media (audio/images/video etc) requires more data.
- Power is required to compress data
- Power is also required to transmit data (so does compressing data save power overall?)
- Transmit power can be saved by error correction, which lets us lower the energy per bit
- Additional data is required for error correction since it requires redundancy
- Power is again required for error correction
- With error correction and data compression, more effective algorithms are usually more complex and power intensive.

To sum up these factors then, we want to increase the capacity of radio channels and use data compression (which reduces the bit rate, obviously) so that we can use Error Correction Coding (ECC, which increases the bit rate) to reduce the amount of transmission power we need and therefore use less power and cause less interference with neighbouring cells.

Furthermore, ECC lets us recover from errors, and we can interleave blocks of data so that bursty errors from radio interference can be mapped onto single errors distributed throughout the blocks. We can gracefully handle un-correctable errors by interpolating between values to minimise the loss of quality.

We can adjust the power of the transmitter based on feedback from the receiver. We must do this constantly, since the user may move around or the environment may change which will affect the quality of the received signal (e.g. from reflections off buildings, doppler shift if you're moving, antenna orientation etc). The goal is to maintain a correctable error rate at the receiver.

6 Medium Access Control

If there are N independent stations each generating packets for transmission at essentially random times on a single channel, then we may have collisions between packets. If we assume that when two packets collide, the contents will be garbled, then our goal should be to reduce the rate of collisions so that packet loss is minimised.

We could use slotted or continuous time when deciding when to transmit packets; slotted time is when the time is divided into fixed slots that transmitters can transmit in. Continuous time means that transmitters can start to transmit packets at any time, not just at the start of the slot boundaries.

Carrier Sense is the term used to describe when a station can detect that the channel is currently busy before it starts to transmit (hence avoid having to re-send the packet again later when it was garbled this time).

We've already looked at the techniques used to allow multiple users to use the same network on mobile phones (FDMA, TDMA, CDMA, OFDMA etc), but one rather older technique called ALOHA has a different approach entirely.

6.1 ALOHA

Pure ALOHA is when stations (this was the 70's) transmit packets whenever they have data. Success is detected by listening to the same channel that they transmit on, or by the receiver acknowledging the frame. If a frame is destroyed, then it is re-transmitted after a random delay. This is known as a **contention** system.

Random delays, of course, avoid the repeated collisions that would occur if the transmitters just kept transmitting sequentially after one frame had failed.

Since if the frame overlaps with any part of another frame, then the vulnerable period for the frame is 3× the length of the frame (see Figure 21).

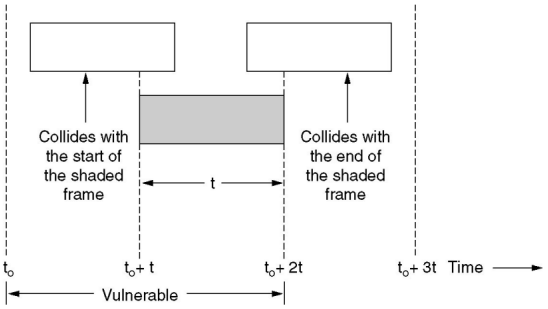


Figure 21: The vulnerable period for the shaded frame is 3× its length.

6.1.1 Slotted ALOHA

If we divide the time domain into slots, then fewer collisions will occur.

Figure 22 shows the difference in efficiency between pure and slotted ALOHA.

6.2 Carrier Sense Multiple Access

This is when a transmitter will check nobody is 'talking' on a channel before it transmits. It will then either:

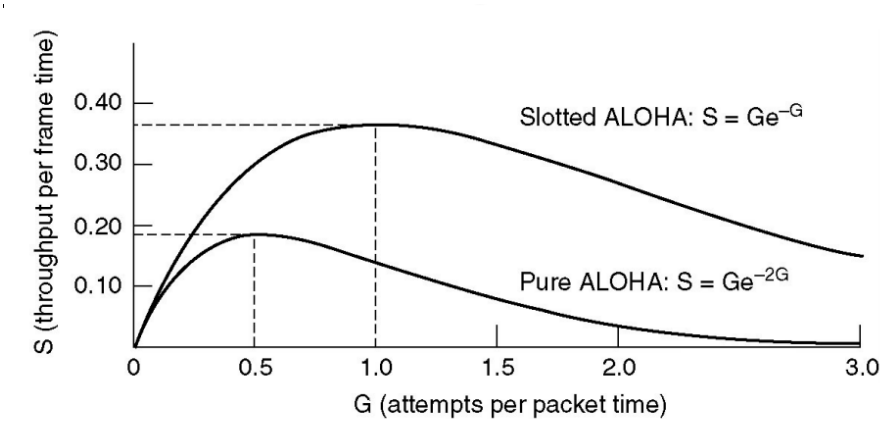


Figure 22: Pure ALOHA achieves 18% utilisation max, whereas Slotted ALOHA gets to 37%.

1-persistent CSMA:
This is when the transmitter waits until the current frame ends before trying to send its own frame.

p-persistent CSMA:
The transmitter might speak at the next slot, the probability that it will do is given by the probability p.

Non-persistent CSMA:
If another transmitter is sending a frame, wait a random time before trying again.

Figure 23 shows a comparison between the different types of CSMA.

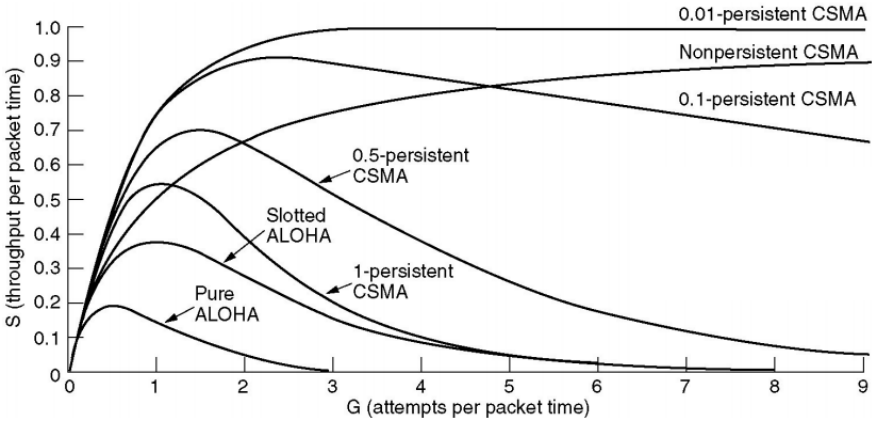


Figure 23: A comparison of channel utilisation versus load for various random access ALOHA protocols.

From the image in Figure 23, it would seem that low persistence is the way to go, since the efficiency is very high. However, if the probability of resending a frame is so low, then it might take ages for the frame to be sent, and so although the network is efficient overall, for any individual, it is very slow.

6.2.1 CSMA collision detection

The network stack is a series of protocols for transmitting data with different levels of abstraction at each layer. The physical layer is the lowest level, and is often what we're dealing with on this course. The data link layer is the next one up, after which there are the internet layer, transport layer and the application layer.

We can save power and transmission capacity by having the transmitter monitor its own transmissions, and abort as soon as it detects a collision. This is implemented by Ethernet, and is present at the physical layer of the network stack. The result of the collision detection is used by the Medium Access Control (MAC) sub-layer, which is part of the data link layer. This makes the decisions about when to transmit and when to back off etc.

6.3 Wireless contention

A wireless transmitter cannot monitor its own transmissions, because it can be either transmitting or receiving, but not both concurrently. Even if a wireless transmitter could do that, it wouldn't matter because it only matters what is being received at the receiver, which is probably different from what is observed at the transmitter.

Because of this, wireless protocols place an emphasis on avoiding collisions:

The whole `RTS/CTS/ACK` thing is called MACA - Multiple Access with Collision Avoidance.

Real channel sensing:
Sense the channel and see if it's free, start transmitting if so. If the receiver senses a transmission, then request a re-transmission after a back-off period.

Virtual channel sensing:
If a device wants to transmit, then it sends a short `RTS` (request to transmit) control frame. If the receiver is ready, it sends a `CTS` (clear to send) frame. The sending device then transmits the message and starts an `ACK` timer. When the receiving device has received the frame, then it will send an `ACK`, and if the sender doesn't receive the `ACK` within the timer period, then it will re-transmit. When other devices hear an `RTS` or `CTS`, they won't transmit for a period of time.

A `NAV` (Network Allocation Vector) flag is one that tells a device to assume the channel is busy and is set whenever a `RTS` or `CTS` frame is heard.

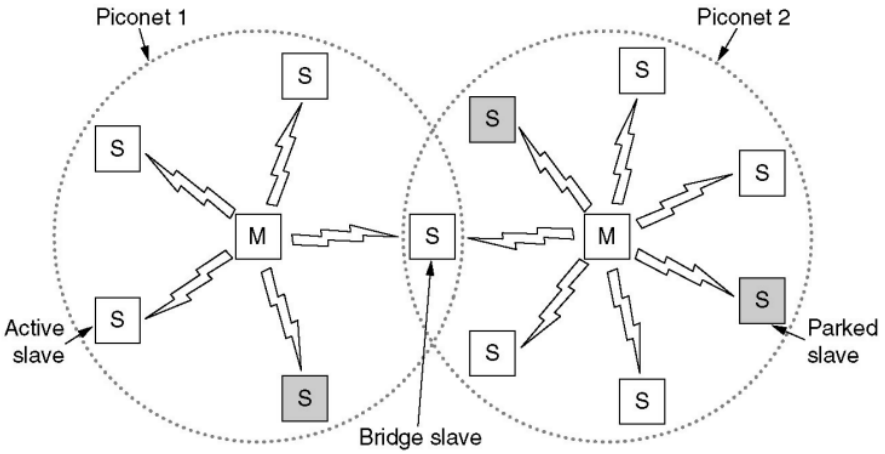


Figure 24: A bluetooth scatternet.

6.3.1 Wireless problems

Wireless comms have some interesting problems that wired comms don't have, mainly to do with the range of the radio:

Hidden Device Problem:
If A wants to transmit to B, but A is too far from C to hear any of C's messages, then it won't be able to hear `RTS` from C. Luckily, B will ignore A's `RTS` until it is finished talking to C.

Exposed Device Problem:
See the lecture notes for this one (lecture 7, slide 18).

6.3.2 Bluetooth

The base unit of a Bluetooth system is a pico-net. This consists of a master node, and up to seven slaves within ten meters. You can connect piconets together to form a 'scatternet' as shown in Figure 24.

The bluetooth designers decided to set a list of things that you could do over Bluetooth, such as file transfers, fax etc. This is in contrast to other protocols such as WiFi, where its up to the user/programmer to think of stuff to do. This makes Bluetooth more rigid.

7 Real time computation

Often, we want to compute something quickly, but the definition of 'quickly' changes depending on the context. Anti-lock breaking, and streaming media obviously have different requirements for real time computation; if media isn't processed fast enough, then you may have a few milliseconds of fuzzy music, but if your anti-lock breaks don't compute fast enough, then you'll probably crash you car.

Henceforth, there are two types of real time computation:

Hard real time:
This is when a missed deadline is a *system failure*.

Soft real time:
This is when a missed deadline is a *loss of quality*.

Some requirements of real time include:

- An event should not occur until a given time has elapsed.
- An event must take place before a given time has elapsed.
- A certain computation must take place within a specified period of time.
- Responding to one event must take priority over responding to another (less important) event.

7.1 Scheduling

Scheduling is how you decide what task to execute next given a list of tasks, the readiness of them, how long they each take to execute and their respective deadlines.

7.2 External events

We could respond to external events simply by polling through each input, but this wastes a lot of power.

An improvement on this is to use interrupts, which is when an external device sends an event flag to the processor so that it switches its attention to handling the event, which incurs context-switch overheads.

Better still would be to use Direct Memory Access (DMA) hardware, which is a separate processor that can move data from external sensors directly into memory without the need for the CPU to do anything.

7.3 Buffering

If the computer needs to process inputs and outputs in real time, then this is very hard. One process simply cannot do two things at once very easily (though it can do things *psudo* at once, we want real time).

A solution to this is to use input and output buffers so that timing can be met precisely, see Figure 25.

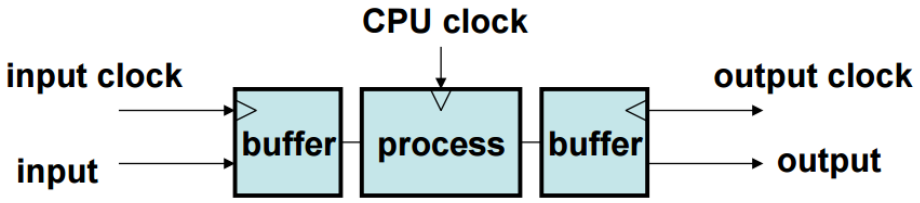


Figure 25: Buffers can allow the CPU to process data in blocks which is more efficient than doing one input at a time.

If we use buffers between the process and the IO, then we can process data in blocks, which is more efficient than doing one input at a time since we can make use of things like parallel instructions to speed things up.

In practice, we might give the buffers over to a DMA controller, which will mean the CPU can merely read and write to memory, and the DMA will take control of all the IO (see Figure 26).

Double buffers allows the CPU to work on one buffer, while the DMA reads another to the output device, and for them to swap over once they have both exhausted their own buffers.

7.4 Timing

Most processor systems incorporate one (or more) timers or counters, often run of a system clock. They can be programmed to interrupt after a certain time, or at regular intervals. This can generate output events at precise times and provides the processor with a reference to real time.

If you've done `COMP22712` then you'll be very familiar with timers in this sense, since you will probably have used one for stuff like waiting n milliseconds, or to drive the processor to check for keyboard presses etc.

Watchdog timers can be used to make sure a system hasn't crashed. A watchdog timer must be reset regularly after a fixed time before it triggers, otherwise it will reset the main CPU. This has the advantage that if the CPU does crash, then real-time functionality can be restored after a brief loss of performance.

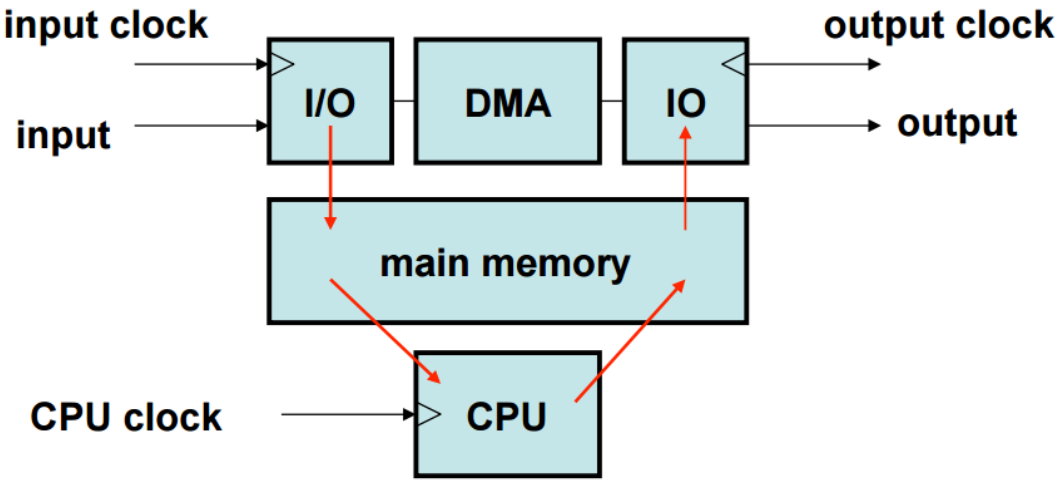


Figure 26: A DMA device interfacing with IO

7.5 Real time communications/Operating Systems

Sometimes, we need to send real time data streams over unreliable and variable delay networks. The Real-time Transport Protocol is often used to facilitate this. If a packet is dropped, then by the time it is resent, it'll be too late, so it is forgotten about. Packets are numbered and timestamped so that out of order packets can be detected and dealt with. Because of the real time requirement, no flow control, error control, acknowledgement or retransmission requesting is implemented, but lost packets can be approximated using interpolation.

Interpolation is a method of constructing new data points within the range of a discrete set of known data points.

In a real time system, if nothing is going on then there is no point computing anything; there are no bonus points for finishing anything early! When there are no events to process, the CPU/network controller etc should power down and disable the clocks. When an event does happen, the CPU can be woken up. This is an event-driven model of computation.

Real Time Operating Systems (RTOS) have event driven priority scheduling, where a task is switched to only when an event of higher priority needs to be processed. Tasks can either be running, ready or blocked, where most tasks are blocked most of the time.

One major issue in real time computing is that context switching (going from one process to another) takes time. A context switch requires you to save the state (registers, open files etc) and load the state of the new one. This can take from tens to hundreds of CPU cycles. Context switches from interrupts are often optimised by using dedicated hardware resources (such as special interrupt registers), which makes it cheaper.

A RTOS can reduce the cost of context switching by using co-routines. These let processes share a stack and other state, and use cooperative multitasking so that a task can hand over control to a higher priority task at a mutually convenient time. This requires programming restrictions to be stringently observed though.

8 Chip Design

In 1948, the Baby was created; a computer that could process 700 instructions per second, using 3.5kW of power. In 2011, the SpiNNaker chip, using an ARM CPU can calculate $200,000,000$ per second, but draws just 40mW of power. This is:

$$\frac{3500}{700} \div \frac{0.04}{200000000} = 25,000,000,000 \text{ times better!}$$

8.1 Jevon's paradox

In 1865, James Watt created a new coal fired steam engine that was much more *efficient* than the current best competitor, and coal consumption *rose* as a result.

It turns out that when technological progress increases the efficiency with which a resource is used, the rate of consumption of the resource usually increases rather than decreases. This is because the efficiency gains usually increase the amount of work done, since people will be spend the same money on the resource as they did before, just get more out of their investment. This in turn, means that the investment makes more sense (since the resource goes further) and demand rises for the resource.

8.2 Low power design

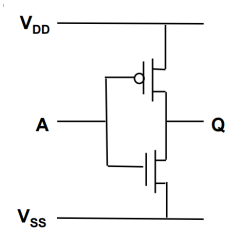
So, why are we still trying to create chips that will operate on lower and lower power in the face of Jevon's paradox? First of all, we want mobile devices to have the maximum battery life we can give them. Furthermore, in the high performance computing industry, we are currently limited by the power required to run the supercomputers. We want lower powered chips that will let us perform more computation for the same (or lower) power cost.

As always with this course, the mobile phone analogy is the best one to use when highlighting the some of the reasons for a trade off. In this case, having a lower power processor means:

- More performance and functionality if we have a faster chip with the same power.
- Using the same amount of power, we could use more CPU cycles for error correction coding.
- If we have more ECC, we can reduce the radio power and save battery life!
- This gives us a market advantage.

8.2.1 CMOS

CMOS stands for Complementary Metal Oxide Semiconductor. This was a revolution in the 60's, since it allowed chip designers to drastically reduce the amount of power used by chips. The reason for this, is that CMOS chips only require power when the voltage on a gate is changing, i.e. when it's switching states. If your CMOS chip isn't doing much, and its gates aren't changing, then it won't use much power at all!



A CMOS inverter, (NOT gate). Note that P is given in Watts.

The power consumed by a gate is equal to:

$$P = \frac{1}{2} \times f_{clock} \times V_{DD}^2 \times \sum \alpha_g C_g$$

Where:

f_{clock}	= Frequency of the clock
V_{DD}	= Supply voltage to processor
C_g	= Capacitance on a gate g
α_g	= The activity on a gate g, which is the mean number of state changes per clock cycle

We can simplify this a bit more by removing the sum, and instead using the total capacitance and average activity. This gives us the power consumed by the chip:

$$P = \frac{1}{2} \times C_{total} \times f_{clock} \times V_{DD}^2 \times \alpha$$

Since the energy used by the chip is equivalent to the power multiplied by the time in seconds that it is used for, we have to worry about that too. If we use too much energy, the battery won't last long, and if we use too much power, then the circuit will heat up and possibly cause damage.

If we want to reduce the power consumption, we can look at optimising the 'large' terms first, in our case V_{DD}^2. If we decrease that, then we will also decrease P. However, if we do this, then each gate will have a lower voltage (of course), which will cause it to switch more slowly, and our critical paths will

take longer to execute. This means we will need to reduce f_{clock} and programs will take longer to run. However, we can use parallelism to offset the increases in circuit delay.

Note that reducing the power consumption of a chip by decreasing f_{clock} doesn't make it more efficient; it still takes the same amount of energy to per instruction. We can also reduce C_{total}, by using simpler circuits (such as ARM instead of x86), and using on chip rather than on chip memory.

Finally, another obvious optimisation is to reduce α. We can do this by turning parts of the circuit off when not in use (for example, don't sit in a wait loop, use interrupts to wake up a processor when its needed), and avoiding distributing the clock signal more than is necessary. 'Event driven' styles of design are good, since they only require something to happen (and thus switching to occur) when there is an event.

9 Code

```
public class HammingCode {

  private final HammingNumber[] numbers;
  int correctionDistance = Integer.MAX_VALUE;

  public HammingCode(HammingNumber[] numbers) {
    this.numbers = numbers;
    for(int i = 0; i < numbers.length - 1; i++) {
      for(int j = i + 1; j < numbers.length; j++) {
        int distance = numbers[i].hammingDistance(numbers[j]);
        if(distance < correctionDistance) {
          correctionDistance = distance;
        }
      }
    }
  }

  public HammingNumber correct(HammingNumber input) {
    for(HammingNumber number : numbers) {
      int distance = input.hammingDistance(number);
      if(distance == 0 || distance < correctionDistance) {
        return number;
      }
    }
    return null;
  }

  public static class HammingNumber {

    private final boolean[] number;

    public HammingNumber(boolean[] number) {
      this.number = number;
    }

    public int hammingDistance(HammingNumber other) {
      int distance = 0;
      for(int i = 0; i < number.length; i++) {
        if(number[i] != other.number[i]) distance++;
      }
      return distance;
    }

    public String toString() {
      StringBuilder sb = new StringBuilder();
      for(boolean b : number) sb.append(b ? "1" : "0");
      return sb.toString();
    }
  }
}
```