Rock-solid diskettes through error correcting codes

Thanassis Tsiodras, Dr.-Ing

October 24, 2001

1 Introduction

I make a living writing code. There is no simpler way of putting it. And living a life messed up with bits and bytes, I am bound to face the inevitable need to transfer them. Be it from work to home, from the company labs to the client's machines or just copying them for backup purposes, the time comes when I have to move them somehow.

In the Internet age, these transfers are mostly done through e-mail attachments or ftp/www servers. Which means that both source and destination machines must have access to the Web.

What do you do when this isn't the case?

You could burn CD-Rs for your files - even though most of the time, your data amounts to less than 5 MB (assuming of course that you have the neccesary equipment inside your computer). If you don't (like me), you could kindly ask your company's sysadmin to do it for you. But what if this happens on a daily basis? You can't constantly bother your sysadmin (unless you want to suddenly see that your computer's web access became rather... slow :). So you do the brave thing, you just compress your files in a couple of disks, and take them with you. Right?

Sadly, this world is governed by Murphy's law. 9 times out of 10, one of your diskettes will have a defective sector or two, which ruins the whole set. Even if you use archivers that employ some kind of error correction (like *rar*, from Eugene Roshal) you basically cross your fingers and hope for the best.

Until now.

2 Algorithm

To be honest, I had this idea about strengthening diskettes for more than a couple of years, but as always, there never was enough time to code this.

The basic plot is that with error correcting codes, like for example **Reed-Solomon**, one can use *parity* bytes to protect a block of data. For my needs, I chose a block of 232 bytes, added 24 bytes of parity (totaling approximately 10% of excess information), and was able to correct up to 12 erroneous bytes inside the 232-byte block. You'd argue of course that a diskette is a block device, that works or fails on 512-byte sector boundaries. True. So all we have to do, is to simply interleave the stream inside the diskette's sectors, in the following way:

1st byte of stream	first byte of first sector
2nd byte of stream	first byte of second sector
...	...
2879th byte of stream	first byte of last sector
2880th byte of stream	second byte of first sector
2881st byte of stream	second byte of second sector
...	...

This way, if a sector is defective, you only loose **one** byte inside your 232-byte block for each of the 512 blocks that pass through that sector - and the algorithm can handle 12 of those errors per block! Taking into account the fact that sector errors are local events (in terms of diskette space), an even better way to interleave is this:

1st byte of stream	first byte of first sector
2nd byte of stream	first byte of Nth sector
...	...
(2880/N) byte of stream	first byte of second sector
((2880/N)+1) byte of stream	first byte of (2+N)th sector
...	...

...where N is a parameter (in my test implementation, I used N=8).

3 Results

Since we use sectors directly, there is of course no concept of a filesystem on the diskette (unless you regard my scheme as one - lets name it RockFAT :). By the way, utilizing this scheme you can use diskettes with bad sectors at the beginning, which are quite useless with FAT ("Track 0 bad - Disk unusable" is what you get when formating such disks). Anyway, we use 232 out of 256 bytes for data, so we have *2880 sectors X 512 bytes X (232/256) = 1336320* bytes available. In my test implementation, I also reserved 20 bytes at the beggining of the stream for a 4-byte stream length, and a 16-byte filename.

So, all you have to do when transfering your files, is to instruct your compression utility (**rar** or **arj** or whatever) to create 1336300-byte blocks. You then transfer these files to your diskettes in the way I described, and you can stop worrying about bad sectors - if it happens, you'll still get your data, at a very high probability[1].

4 Conclusion

Don't throw away your "Track 0 bad" diskettes! Don't pest your company's sysadmin! Just use RockFAT :) Seriously, this idea can obviously be implemented for other mediums - even be a true filesystem, in the ext2 and NTFS sense. Go ahead and implement it to your heart's desire.

A sample implementation for Windows (and Linux, through `dd` usage) is available at my site, `http://www.softlab.ntua.gr/~ttsiod/`. Don't expect a GUI though :) It works as is for me, hope it does for you too...

[1] To loose data, your diskette must have 12 bad sectors on 8-sector intervals, for example: sectors 0,8,16,..,96 must be bad. Quite unlikely.