The debdelta suite

Andrea C. G. Mennucci

debdelta is an application suite designed to compute changes between Debian packages. These changes (that we will call 'deltas') are similar to the output of the "diff" program in that they may be used to store and transmit only the changes between Debian packages. This suite contains 'debdelta-upgrade', that downloads deltas and use them to create all Debian packages needed for an 'apt-get upgrade'.

1. Overview

The debdelta application suite is really composed of different applications.

1.1. debdelta

debdelta computes the delta, that is, a file that encodes the difference between two Debian packages. Example:

```
$ a=/var/cache/apt/archives
$ debdelta -v $a/emacs-snapshot-common_1%3a20060512-1_all.deb \
$a/emacs-snapshot-common_1%3a20060518-1_all.deb /tmp/emacs.debdelta
```

the result is: deb delta is 12.5% of deb ; that is, 15452kB would be saved

1.2. debpatch

debpatch can use the delta file and a copy of the old Debian package to recreate the new Debian package. (This process is called "applying the delta file"). If the old Debian package is not available, but is installed in the host, it can use the installed data; in this case, '/' is used in lieu of the old .deb.

Example:

```
$ debpatch -A  /tmp/emacs.debdelta / /tmp/emacs.deb
```

1.3. debdeltas

debdeltas can be used to generate deltas for many debs at once. It will generate delta files with names such as `package_old-version_new-version_architecture.debdelta`. If the delta exceeds ~70% of the deb, 'debdeltas' will delete it and leave a stamp of the form `package_old-version_new-version_architecture.debdelta-too-big`. Example usages are in the man page; see also Section 3.9.

1.4. debdelta-upgrade

debdelta-upgrade will download necessary deltas and apply them to create debs for a successive **apt-get upgrade**. The deltas are available for upgrades in 'stable' , 'stable-security' , 'testing', 'unstable' and 'experimental', for i386 and amd64. Example usage:

```
# apt-get update && debdelta-upgrade && apt-get upgrade
```

If run by a non-root user, debs are saved in /tmp/archives : do not forget to move them in `/var/cache/apt/archives`

debdelta-upgrade will also download .debs for which no delta is available (this is done in parallel to patching, to maximize speed). See the explanation of "debdelta-upgrade --deb-policy" in the man page for more informations and customization on which debs get downloaded.

More informations are in next sections.

1.5. debforensic

There is also another bunch of code (that though was never distributed.... it is available in the GIT repo). . **debforensics** creates and uses sqlite databases containing information regarding debian binary packages. **debforensics --add** will scan debian packages and add the list of files (and SHA1 hashes of them) to the database. **debforensics --scan** will check a file against multiple databases, to see if that file is part of any package. **debforensics --forensic** will scan a filesystem and list files that are part of a package, and files that are not (or are missplaced, or have strange permissions....).

If debdelta-upgrade fails to apply a delta, and '-d' is passed, then a debug file is generated, and then debforensic may be used to understand what went wrong (theoretically).

> **Important:** Beware: a full database for main/amd64 is ~350MBs, without indexes. So in practice currently I cannot keep a database in my host.

2. a delta

The delta is 'ar' archive (see 'man ar'). The delta contains 'info', some data members (named by numbers), a script named 'patch.sh.xxx', and optional gpg signatures. The script recreates the new deb. See `do_delta_()` in the python code for more details.

2.1. the info in a delta

a delta first 'ar' member is always named 'info', and is a text file containing some keywords and informations regarding the delta itself. [TODO add details]

2.2. how to apply a delta

TODO WRITEME. You may look into `/usr/share/debdelta/debpatch.sh` to understand the basics.

3. debdelta-upgrade service

In June 2006 I set up a delta-upgrading framework, so that people may upgrade their Debian box using **debdelta-upgrade** (that downloads package 'deltas'). This section is an introduction to the framework that is behind 'debdelta-upgrade', and is also used by 'cupt'. In the following, I will simplify (in places, quite a lot).

3.1. The framework

The framework is so organized: I keep up some servers where I use the program 'debdeltas' to create all the deltas; whereas endusers use the client 'debdelta-upgrade' to download the deltas and apply them to produce the debs needed to upgrade their boxes. In my server, I mirror some repositories, and then I invoke 'debdeltas' to make the deltas between them. I use the scripts `/usr/share/debdelta/debmirror-delta-security` and `/usr/share/debdelta/debmirror-marshal-deltas` for this. This generates any delta that may be needed for upgrades in squeeze,squeeze-security,wheezy,sid,experimental, for architectures i386 and amd64 (as of Mar 2011); the generated repository of deltas is more or less 10GB.

3.2. The goals

There are two ultimate goals in designing this framework:

1. SMALL) reduce the size of downloads (fit for people that pay-by-megabyte);
2. FAST) speed up the upgrade.

The two goals are unfortunately only marginally compatible. An example: bsdiff can produce very small deltas, but is quite slow (in particular with very large files); so currently (2009 on) I use 'xdelta3' as the backend diffing tool for 'debdeltas' in my server. Another example is in debs that contain archives (.gz, , tar.gz etc etc): I have methods and code to peek inside them, so the delta become smaller, but the applying gets slower.

3.3. The repository structure

The repository of deltas is just a HTTP archive; it is similar to the pool of packages; that is, if `foobar_1_all.deb` is stored in `pool/main/f/foobar/` in the repository of debs, then the delta to upgrade it will be stored in `pool/main/f/foobar/foobar_1_2_all.debdelta` in the repository of deltas. Contrary to the repository of debs, a repository of deltas has no indexes, see Section 3.7.2. The delta repository is in `http://debdeltas.debian.net/debian-deltas`.

3.4. The repository creation

Suppose that the unstable archive, on 1st Mar, contains `foobar_1_all.deb` (and it is in `pool/main/f/foobar/`) ; then on 2nd Mar, `foobar_2_all.deb` is uploaded; but this has a flaw (e.g. FTBFS) and so on 3rd Mar `foobar_3_all.deb` is uploaded. On 2nd Mar, the delta server generates `pool/main/f/foobar/foobar_1_2_all.debdelta` On 3rd Mar, the server generates both `pool/main/f/foobar/foobar_1_3_all.debdelta` `pool/main/f/foobar/foobar_2_3_all.debdelta`. So, if the end-user Ann upgrades the system on both 2nd and 3rd Mar, then she uses both foobar_1_2_all.debdelta (on 2nd) and `foobar_2_3_all.debdelta` (on 3rd Mar). If the end-user Boe has not upgraded the system on 2nd Mar, , and he upgrades on 3rd Mar, then on 3rd Mar he uses `foobar_1_3_all.debdelta`.

3.5. size limit

Note that currently the server rejects deltas that exceed 70% of the deb size: indeed the size gain would be too small, and the time would be wasted, if you sum the time to download the delta and the time to apply it (OK, these are run as much as possible in parallel, yet ....).

Also, the server does not generate delta for packages that are smaller than 10KB.

3.6. /etc/debdelta/sources.conf

Consider a package that is currently installed. It is characterized by *name installed_version architecture* (unfortunately there is no way to tell from which archive it came from, but this does not seem to be a problem currently) Suppose now that a newer version is available somewhere in an archive, and that the user wishes to upgrade to that version. The archive Release file contain these info: “Origin , Label , Site, Archive”. (Note that Archive is called Suite in the Release file). Example for the security archive:

```
Origin=Debian
Label=Debian-Security
Archive=stable
Site=security.debian.org
```

The file `/etc/debdelta/sources.conf` , given the above info, determines the host that should contain the delta for upgrading the package. This information is called "delta_uri" in that file. The complete URL for the delta is built adding to the delta_uri a directory path that mimicks the "pool" structure used in Debian archives, and appending to it a filename of the form `name_oldversion_newversion_architecture.debdelta`. All this is implemented in the example script contrib/findurl.py . If the delta is not available at that URL, and `name_oldversion_newversion_architecture.debdelta-too-big` is available, then the delta is too big to be useful. If neither is present, then, either the delta has not yet been generated, or it will never be generated... but this is difficult to know.

3.7. indexes

3.7.1. indexes of debs in APT

Let’s start examining the situation for debs and APT. Using indexes for debs is a no-brainer decision: indeed, the client (i.e. the end user) does not know the list of available debs in the server, and, even knowing the current list, cannot foresee the future changes. So indexes provide needed informations: the packages’ descriptions, versions, dependencies, etc etc; these info are used by apt and the other frontends.

3.7.2. no indexes of deltas in debdelta

If you then think of deltas, you realize that all requirements above fall. Firstly there is no description and no dependencies for deltas. [1] Of course ’debdelta-upgrade’ needs some information to determine if a delta exists, and to download it; but these information are already available:

```
the name of the package P
the old version  O
the new version  N
the architecture A
```

Once these are known, the URL of the file F can be algorithmically determined as `URI/POOL/P_O_N_A.debdelta` where URI is determined from `/etc/debdelta/sources.conf` and POOL is the directory in the pool of the package P . This algorithm is also implemented (quite verbosely) in contrib/findurl.py in the sources of debdelta. This is the reason why currently there is no "index of deltas", and nonetheless 'debdelta-upgrade' works fine (and "cupt" as well). Adding an index of file would only increase downloads (time and size) and increase disk usage; with negligeable benefit, if any.

3.8. no incremental deltas

Let me add another point that may be unclear. There are no incremental deltas (and IMHO never will be).

3.8.1. What "incremental" would be, and why it is not

Please recall Section 3.4. What *does not happen* currently is what follows: on 3rd Mar , Boe decides to upgrade, and invokes 'debdelta-upgrade'; then 'debdelta-upgrade' finds `foobar_1_2_all.debdelta` and `foobar_2_3_all.debdelta` , it uses the foremost to generate `foobar_2_all.deb`, and in turn it uses this and the second delta to `generate foobar_3_all.deb` . This is not implemented, and it will not, for the following reasons.

- The delta size is, on average, 40% of the size of the deb (and this is getting worse, for different reasons, see Section 5.2); so two deltas are 80% of the target deb, and this too much.
- It takes time to apply a delta; applying two deltas to produce one deb takes too much time.
- The server does generate the direct delta `foobar_1_3_all.debdelta` :-) so why making things complex when they are easy? :-)
- Note also that incremental deltas would need some index system to be implemented... indeed, Boe would have no way to know on 3rd Mar that the intermediate version of foobar between "1" and "3" is "2"; but since incremental deltas do not exist, then there is no need to have indexes).

3.9. Repository howto

There are (at least) two ways two manage a repository, and run a server that creates the deltas

3.9.1. debmirror --debmarshal

The first way is what I currently use. It is implemented in the script `/usr/share/debdelta/debmirror-marshal-deltas` (a simpler version, much primitive but more readable , is `/usr/share/debdelta/debmirror-delta-security`) Currently I use the complex script that creates deltas for amd64 and i386, and for lenny squeeze sid experimental ; and the simpler

one for lenny-security. Let me start outlining how the simple script generate deltas . It is a 3 steps process. Lets say that $secdebmir is the directory containg the mirror of the repository security.debian.org.

```
1. --- 1st step
 #make copy of current stable-security lists of packages
 olddists=${TMPDIR:-/tmp}/oldsecdists-`date +'%F_%H-%M-%S'`
 mkdir $olddists
 cp -a $secdebmir/dists $olddists
```

2. --- 2nd step call 'debmirror' to update the mirror ; note that I apply a patch to debmirror so that old debs are not deleted , but moved to a /old_deb directory
3. --- 3rd step call 'debdeltas' to generate deltas , from the state of packages in $olddists to the current state in $secdebmir , and also wrt what is in stable. Note that, for any package that was deleted from the archive, then 'debdeltas' will go fishing for it inside /old_deb .

The more complex script uses the new *debmirror --debmarshal* so it keeps 40 old snapshots of the deb archives, and it generates deltas of the current package version (the "new" version) to the versions in snapshots -10,-20,-30,-40.

3.9.2. hooks and repository of old_debs

I wrote the scheleton for some commands.

debdelta_repo [--add name version arch filename disttoken]

This first one is to be called by the archive management tool (e.g. DAK) when a new package enters in a part of the archive (lets say, package="foobar" version="2" arch="all" and filename="pool/main/f/foobar/foobar_2_all.deb" just entered disttoken="testing/main/amd64"). That command will add that to a delta queue, so appropriate deltas will be generated; this command returns almost immediately.

debdelta_repo [--delta]

This does create all the deltas.

debdelta_repo [--sos filename]

This will be called by DAK when (before) it does delete a package from the archive; this command will save that old deb somewhere (indeed it may be needed to generate deltas sometimes in the future). (It will be up to some piece of *debdelta_repo* code to manage the repository of old debs, and delete excess copies).

TODO that scheleton does not handle 'security', where some old versions of the packages are in a different DISTTOKEN

4. Goals, tricks, ideas and issues

4.1. exact patching

When **debpatch** or **debdelta-upgrade** recreates a .deb, it will be identical to the desired one (so it may be possible to check it using the security features in APT (http://wiki.debian.org/SecureApt) [2]). See though Section 4.5.

4.2. exact recompression

Suppose a .deb has inside a huge file /usr/share/doc/foobar/document.info.gz and this starts with a RCS tag ... then each time it is released, the file will be different even though just few bytes were changed. Another examples are manpages that start with the header containing the version of the command. So , to get good compression of the difference, I had to be able to gunzip those files, diff them, and gzip back them *exactly identical* (but possibly for headers [3]) For this reason, I studied gzip formats, and I wrote in debdelta some python code that does the trick (90% of the times...). [4]

4.3. speed

4.3.1. some (old) numbers

Warning: this section is referred to experiments done in 2006, and the backend for delta encoding was 'xdelta'. On a desktop with CPU Athlon64 3000 and a average hard disk,

```
$ debdelta mozilla-browser_1.7.8-1sarge3_i386.deb \
mozilla-browser_1.7.8-1sarge6_i386.deb /tmp/m-b.debdelta
```

processes the 10Mb of mozilla-browser in ~11sec, that is a speed of ~900kB per second. Then debpatch applies the above delta in 16sec, at a speed of ~600kB per second. Numbers drop in a old PC, or in a notebook (like mine, that has a Athlon 1600MHz and slow disks), where data are chewed at ~200kB per second. Still, since I have a ADSL line that downloads at max 80kB per second, I have a benefit downloading deltas. In a theoretical example, indeed, to download a 80MB package, it would take 1000seconds; whereas to download a delta that is 20% of 80MB it takes 200seconds, and then 80MB / (200kB/sec) = 400seconds to apply it, for a total of 600seconds. So I may get a "virtual speed" of 80MB / 600sec = 130kB/sec . Note that delta downloading and delta patching is done in parallel: if 4 packages as above have to be downloaded, then the total time for downloading of full debs would be 4000seconds, while the time for parallel-download-patch-apply-patch may be as low as 1400seconds.

This is a real example of running 'debdelta-upgrade' :

```
Looking for a delta for libc6 from 2.3.6-9 to 2.3.6-11
Looking for a delta for udev from 0.092-2 to 0.093-1
```

```
Patching done, time: 22sec, speed: 204kB/sec, result: libc6_2.3.6-11_i386.deb
Patching done, time: 4sec, speed: 57kB/sec, result: udev_0.093-1_i386.deb
Delta-upgrade download time 28sec speed 21.6k/sec
total time: 53sec; virtual speed: 93.9k/sec.
```

(Note that the "virtual speed" of 93.9k/sec , while less than the 130kB/sec of the theoretical example above, is still more than the 80kB that my ADSL line would allow). Of course the above is even better for people with fast disks and/or slow modems. Actually, an apt delta method may do a smart decision of how many deltas to download, and in which order, to optimize the result, (given the deltas size, the packages size, the downloading speed and the patching speed).

4.3.2. speeding up

The problem is that the process of applying a delta to create a new deb is currently slow, even on very fast machines. One way to overcome is to "parallelize as much as possible". The best strategy that I can imagine is to keep both the CPU, the hard disk, and the Internet connection, always maxed up. This is why ’debdelta-upgrade’ has two threads, the "downloading thread" and the "patching thread". The downloading thread downloads deltas (ordered by increasing size), and as soon as they are downloaded, it queues them to be applied in the "patching thread"; whereas as soon as all available deltas are downloaded it starts downloading some debs, and goes on for as long as the deltas are being applied in the "patching thread". Summarizing, the downloading thread keeps Internet busy while the patching thread keeps the CPU and HDD busy.

Another speedup strategy is embedded inside the deltas themselves: since bsdiff is a memory hog, when the backend is bsdiff, I have to divide the data in chunks; this may lower the compression ratio, but the good point is that the HDD accesses and the calls to bsdiff can run "in parallel". With newer xdelta3, xdelta3 can read the original data from a pipe, so the data are not divided in chunks, but rather continously piped into xdelta3; so xdelta3 runs at the same time as when the data are read from HDD.

4.3.3. the 10kb trick

currently, roughly half of the generated deltas[5] are less than 10KB. **debdelta-upgrade** downloads deltas in two passes,

1. in the first pass it tries to download the first 10KB of a delta; if it gets a complete delta, it immediatly pipes it in the "patching thread queue", otherwise if it gets only a partial download, it adds it to the download queue; if it gets HTTP404, it possibly checks for the "toobig" timestamp, and it possibly warns the user.
2. in the second pass, it downloads the rest of the deltas, and queues them for patching

Why this complex method? because the first 10KBs of a delta contain the info, and those may be used to actually decide not to download the rest of the delta (if a TODO predictor decides that it is not worthwhile...Section 4.3.4).

4.3.4. the choice, the predictor

Which deltas should be downloaded, VS which debs? Currently there is a rule-of-thumb: the server immediately deletes any delta that exceeds 70% of the original deb , and it replaces it with an empty file ending in ".debdelta-too-big". In such cases, "debdelta-upgrade" will download the deb instead. See the explanation of "debdelta-upgrade --deb-policy" in the man page for more info and customization on which debs get downloaded.

Some time ago I tried to do devise a better way to understand when to download a delta w.r.t. a deb. The code is in the "Predictor" class .... but I could not reliably predict the final speed of patching, so currently it is not used.

4.3.5. State of the art

All in all, I still cannot obtain high speeds: so people that have a fast ADSL Internet connection usually are better downloading all the debs, and ignoring "debdelta-upgrade" alltogether. Anyway, the best way to know is to try "debdelta-upgrade -v" and read the final statistics. See Section 4.7 and Section 4.8 for recent developments.

4.4. better deb compression is a worse delta

'xdelta3' can reconstruct data at high speed: on nowadays processors, it can process up to 2MB per second; but, when applying a delta, 'xdelta3' works on *uncompressed data*. So if the data is then compressed at a ratio 1/3, then the resulting speed on *compressed data* is 700KB/sec. Moreover, time is needed to actually compress the data.

In recent years, 'dpkg' has transitioned from 'data.tar.gz' to 'data.tar.bz2' to 'data.tar.lzma'; each method is better at compressing, but is also slower than the previous one; since it is better at compressing, it also defeats the ability of 'debdelta' to produce small deltas (wrt the original deb, of course), and indeed statistics show that deltas are getting larger; since it is slower, it slows down the applying of deltas as well.

4.5. long time recovery

As aforementioned, deltas can rebuild the deb identically to the byte. But the patch.sh script calls the standard tools 'tail','head','zgip','bzip2','lzma', etc etc to rebuild a delta; so if the argument calling or output of any of those tools changes, than a delta may become unusable. As long as deltas are used for the debdelta-upgrade service, this is no big deal: if such a tool changes, then we can adjust the deltas to it, and there is just some days disruption of the service [6] (and people will download debs instead of deltas .... as we used to).

If anybody wants instead to use debdelta to archive debs for long time, (as the archive.debian.org service was doing), then we should make sure that , at any moment in future, deltas can be applied. A possible solution would be that deltas should contain, in the info files, the versions of all tools that are needed for applying. A second solution is that debdelta should keep a standard set of those tools inside the package.

4.6. streaming

Let me summarize. When 'debdelta-upgrade' (or 'debpatch') recreates a deb, one step is reassembling the data.tar part inside it; this part moreover is compressed (gzip, bzip2 or lately lzma). This 'reassembling and compressing' takes time (both for CPU and for HD), and is moreover quite useless, since, in short time, 'apt' will call 'dpkg -i' that decompresses and reopens the data.tar in the deb.

It is then reasonable to collapse this two parts, and this would possibly speed up the upgrade a bit. A first step is *'--format=unzipped'* Section 4.7 , a next step may be *'--format=preunpacked'* Section 4.8.

4.7. --format=unzipped

The recently introduced new `--format=unzipped` may speed up package upgrades. If you call 'debdelta-upgrade' with the option '--format=unzipped' , then in the recreated deb the data.tar part will not be compressed. This may speedup the 'debdelta-upgrade' + 'apt-get upgrade' process. Indeed, writing to hard disk is fast (let's say 5MB/sec, but usually much more); whereas compressing random data with 'bzip2 -9' or 'lzma -9' is much slower (let's say 2.0MB/sec and 1.5 MB/sec) ; and moreover the compressed data is then decompressed by dpkg when installing; so avoiding the compress/decompress should be a win/win (unless you run out of disk space...). Indeed I see that the creation of deltas is much faster; but I still do not have enough data collected....

4.8. --format=preunpacked

Here is another idea. When 'debdelta-upgrade' is called in upgrading a package 'foobar' it currently creates 'foobar_2.deb'. By an appropriate cmdline switch *'--format=preunpacked'*, instead of creating a 'foobar_2.deb' , it directly saves all of its file to the filesystem, and it adds an extension to all the file names, making sure that no file name conflicts (=overwrite) with a preexisting file on the filesystem ; then it creates a file 'foobar_2.deb_preunpacked' , that is a deb package were 'data.tar.xxx' is replaced with 'data_list', just a text file specifying the contents of 'data.tar.xxx' and where regular files were temporarily unpacked.

Note that the above idea overlaps a lot with the SummerOfCode2010 StreamingPackageInstall (http://wiki.debian.org/SummerOfCode2010/StreamingPackageInstall)

debdelta-upgrade --format=preunpacked is now implemented as a proof-of-concept (it does not really write temporary files to HD yet). The format of *data_list* is

```
Files:
 TYPE MODE USER GROUP MTIME
 NAME_FILE_WAS_UNPACKED_TO (if regular file)
 ORIGINAL_FILENAME
 LINK_NAME (if link)
[repeat]
```

Example of data_list

```
Files:
 d 0755 root root 1304626623

 ./etc

 - 0644 root root 1304626594
 /./etc/gnashrc_1956_debdelta_preunpacked
 ./etc/gnashrc
 l 0777 root root 1304626629

 ./usr/share/man/man1/gtk-gnash.1.gz
 gnash.1.gz
```

PROS: (1) may be faster; (2) if you need to upgrade a 100MB package, you do not need to save both the deb and (while 'dpkg --unpack') the whole new deb data : so there is less risk of running our of disk space.

CONS: (1) you cannot install that "preunpacked deb" twice (so dpkg should probably remove it once it has installed it); (2) you cannot move it to another host; (3) when "apt-get clean", all temporary files have to be removed as well.

So it may be a good idea to use ".deb_preunpacked" as extension for them. And I would recommend using '--format=unzipped' for essential packages such as the kernel.

If you like the idea, someone should help in changing 'dpkg' so that it would be able to install starting from 'foobar_2.deb_preunpacked'. And change APT so that it would interact with 'debdelta' to create the 'foobar_2.deb_unpacked' files, and pass them to dpkg (and clean them properly).

5. Todo

5.1. todo list

1. Prepare an APT method so that 'apt-get upgrade' would actually use deltas. Some code is already written. See also 2011 Google Summer of Code.
2. As in Section 4.3.4. It would be nice if debdelta-upgrade would actually choose if
   - download a delta and use it to create the .deb
   - download the deb

   depending on which one would be faster. Unfortunately, this decision must depend on a good model to predict the speed of patching... and this I still cannot achieve.
3. in debdelta-upgrade, have as many "patching thread" as there are cores
4. upgrade debdelta-upgrade to newer libapt
5. support multiarch
6. collect data, benchmark! (some debdelta behaviours are coded in magic numbers that I got from thumb reasoning on small datasets)
7. support long time exact recovery Section 4.5: embed a copy of gzip, libzip, bzip2 and lzma in debdelta??

5.2. things are getting worse

W.r.t. to when I started deploying debdelta, things got worse, for two reasons,

1. one problem is Section 4.4
2. delta backends are bad at compressing a binary that was compiled from the same source but with twi different compilers; see in particular the Google Courgette project (http://dev.chromium.org/developers/design-documents/software-updates-courgette), and compare it with the problems I encountered lately when Debian switched from GCC 4.4 to 4.5, (http://debdelta.debian.net/run/tests/debs-newer-gcc/bibledit/) when it happened that the binaries were so different that the compression of the new binary with LZMA would be smaller than the BSDIFF of the old and the new binary (!!). Unfortunately it seems that Google Courgette was hit with a patent infringment (http://www.h-online.com/open/news/item/Patent-action-over-Google-s-Courgette-845028.html)

so we should study how to reduce the size of deltas, and/or making them faster (possibly implementing lzma in xdelta3; or automatically choosing 'bsdiff' vs 'xdelta3' depending on the situation).

Notes

1. deltas have a "info" section, but that is, as to say, standalone

2. note though that **debdelta-upgrade** saves the recontructed debs in `/var/cache/apt/archives`, and APT does not check them there, AFAICT
3. the re-gzipped files are identical but for headers, (indeed gzip headers contain sometimes a timestamp); but this is not a problem since the reconstructed gzipeed file is then piped again into 'xdelta3' or 'bsdiff' to rebuild the 'data.tar', so the header is fixed at that stage
4. This is implemented in the python routine `delta_gzipped_files`.
5. that is, discarding those that are more than 70% of the corresponding deb
6. this actually already happened some years ago, with libzip