0. Overview

Going over the 3 aspects of CRDS use we discussed at the telecon, here is how I think we should deploy CRDS remotely:

1. **Pipeline Cache Setup** -- Initialize your pipeline CRDS cache by leveraging the existing CDBS reference files. This saves 0.5 – 1.5 T of downloads. The crds.sync tool is used to set up most of the cache and download all the CRDS rules. Ad hoc links or copies set up references. crds.sync can check files.

2. **Pipeline Cache Maintenance and Bestrefs** – Script "cron_sync" runs every 30 minutes for background caching of references, mappings, and hst-operational context update. Script "safe_bestrefs" wraps crds.bestrefs to ensure pipeline and cron concurrency management. safe_bestrefs runs in server-less mode making pipeline runs independent of the CRDS server. Only cron_sync updates the cache.

3. **Affected Datasets** -- The CRDS server runs another cronjob, affected_datasets, to generate a list of datasets affected by a context change, and sends them to a majordomo list as a compressed text file attachment. The affected datasets will be computed whenever the default CRDS context changes and the same list sent to all sites.

1. CRDS Pipeline Cache Setup

1.1 CRDS Environment Vars

A CRDS ops pipeline environment for HST can be set up as follows:

```
% setenv CRDS_PATH   <where you want (1.5T – existing storage) of cache>
% setenv CRDS_SERVER_URL  https://hst-crds.stsci.edu
```

A CRDS test pipeline environment for HST can be set up as follows:

```
% setenv CRDS_PATH   <where you want your test cache>
% setenv CRDS_SERVER_URL  https://hst-crds-test.stsci.edu
```

NOTE: nominally test and ops systems have independent caches and servers. There is no guarantee that the test and ops systems have the same files or behave in exactly the same way... the whole point of test.

Similar URLs do/will exist for JWST but the JWST servers are currently less mature.

Afterward, you should be able to:

```
% python -m crds.list –config
```

and see something about your CRDS cache and environment setup. crds.list is not concurrency-safe and should not be run in an operational pipeline account on a read-write cache.

1.3 CRDS remote cache init

There's an attached script which runs in about 5 minutes to set up the initial CRDS cache. The cache contains all the available CRDS mappings and configuration information but no references.

```
% remote_cache_init
```

1.4 CDBS references linking

Copying all HST references across continents with crds.sync might take a week or two. So I propose a more ad hoc approach of linking your existing CDBS references into your CRDS cache and then touching up afterward. Since I don't know how your CDBS references are organized, this is incomplete and you will have to futz around. Conceptually here's what to do:

```
% foreach reference ( iref/* jref/* ...)
... ln -s ${reference}   ${CRDS_PATH}/references/hst
... end
```

NOTE: ${reference} needs to evaluate to an absolute path. So use absolute iref, jref, etc. in the foreach or doctor ${reference} with the appropriate shell command.

My intent here is that CDBS and CRDS share the same files. If you plan own blowing away the CDBS directory organization later and keep the CRDS organization, use hard links. If you want to play around with this some with less risk to your references, make them readonly, and use symlinks as I've shown. If you want independent reference stores, copy instead of linking.

1.5 CRDS/CDBS reference verification

The crds.sync tool can “verify” a cache by using information from the server. There are two levels of verification: fast and comprehensive. The principle advantage of crds.sync over rsync is that it does not require a UNIX account at STScI to use... and as something which is not a strict “mirror”, supports local variations in the files you retain.

1.5.1 Fast check:

The fast verification just checks the file length, and ensures that it is present.

```
% python -m crds.sync –all –fetch-references –check-files –verbose  |& tee fast_check.log
```

1.5.2 Comprehensive check:

The comprehensive version checks file existence, length, and sha1sum, essentially guaranteeing identical contents to the server/archive version of the file.

```
% python -m crds.sync –all –fetch-references –check-files –check-sha1sum –verbose  |& tee
comprehensive_check.log
```

1.5.3 File repair:

Conservative, the –check options don't fix anything by default. To delete and re-download broken files:

```
% python -m crds.sync –all –fetch-references –check-files –check-sha1sum –repair-files –verbose  |& tee
repair.log
```

In the case of our “hybrid caches”, the downloaded copy does not mutate the original, it creates a new redundant copy in the CRDS cache. The comprehensive check will take hours or days to cover all files with sha1sum.

1.5.4 Removing files:

crds.sync permits us to remove files from the CRDS cache which are no longer wanted, primarily to conserve space. This is complex so I'll assume that institutions are not interested and want to keep all files.

caveat: One area where this may be interesting, even to Institutions, is removing unusable CDBS files from the CRDS cache. Since we're blanket copying your CDBS files, these unusable obsolete files may exist in quantity.

Removing files is controlled with –purge-mappings and –purge-references. To do a non-destructive estimate, add –dry-run, which IIRC only affects purging. The rules and references purged are the ones which are not reachable from the contexts you specify to crds.sync. If you say –all, that means all contexts, so –purge-mappings will do nothing, and –purge-references should only remove non-CRDS files. There are several ways to specify subsets of CRDS contexts, probably more useful for retrieving than purging.

1.6 Conclusion

At this point you should have an initialized CRDS cache. If you choose have possibility of verifying it, either rapidly or bit-for-bit, with the CRDS server. The CRDS server is currently initialized with, and distributes references taken from, /grp/hst/cdbs, not the HST archive. Those files define the current sha1sums.

3. CRDS Pipeline Cache Updates and Concurrent Bestrefs

On-going synchronization is a tricky issue. The CRDS release under test does not fully manage concurrency, so I have created wrapper scripts to give you as part of setup and integration, which will be folded into subsequent CRDS distributions. After some discussion with Mike I'm proposing the following approach:

3.1 cron_sync

A cronjob (cron_sync) runs every 30 minutes to download newly archived references and mappings in the background. The cronjob is a wrapper script around crds.sync which manages concurrency. The run time of the cronjob is unbounded but expected to be in the 10's of minutes to low hours. The cronjob will download files from the CRDS server only when they have been successfully archived; it should be noted that the files do not currently come from the archive. cron_sync fails immediately if it is still/already running. cron_sync should not block bestrefs for more than 30 seconds. The goal is that only cron_sync modifies the CRDS cache.

And example addition to your crontab file looks like:

```
*/30 * * * *   cron_sync --all --fetch-references --verbose -check-files
```

You set a crontab with:

```
% crontab my.crontab.file
```

And check it with:

```
% crontab -l  (-L)
```

And suppress routine e-mail by appending something like:

```
>& /dev/null
```

to the contab entry.

If cron_sync run time performance becomes an issue with –all, more restrictive versions are available which may accomplish what is actually required:

```
*/30 * * * *   cron_sync --contexts hst-operational --fetch-references --verbose -check-files
*/30 * * * *   cron_sync --last 10 --fetch-references --verbose -check-files
```

There's nothing preventing it, but don't run cron_sync with –check-sha1sum or you will block safe_bestrefs (or run it in a less sound way) for potentially hours or days.

3.2 safe_bestrefs

A wrapped version of crds.bestrefs, safe_bestrefs, is run in the pipeline to update dataset headers with best reference recommendations. safe_bestrefs does not block itself. safe_bestrefs manages concurrency between itself and cron_sync. safe_bestrefs blocks but times out after 3 minutes if the cron_sync fails to release crds.config.lock. So this is “imperfect concurrency control in depth”, we try to ameliorate, but don't over-estimate the potential problem when things go wrong, i.e. automatically elevate hypothetical problems to the real deadlocks.

After setting CRDS_PATH, example safe_bestrefs invocation looks like:

```
% setenv CRDS_PATH <your cache>
% safe_bestrefs -new-context hst-operational -files *_raw.fits   --update-bestrefs
```

The context used can be explictly specified to prevent automatic updates for perform special processing:

```
% safe_bestrefs -new-context hst_0042.pmap -files *_raw.fits   --update-bestrefs
```

The context will default to hst-operational if you say nothing:

```
% safe_bestrefs -files *_raw.fits   --update-bestrefs
```

--update-bestrefs is a required parameter, otherwise a dry-run which doesn't change headers will be performed.

5. Affected Data Sets

The CRDS server will run a cronjob which monitors the current operational context every 10 minutes. When the context changes, a mode of crds.bestrefs will be run which does a context-to-context comparison using dataset parameters from DADSOPS, computing a list of dataset ids which are affected by the context change.

The result of the cronjob is an e-mail which contains two attachements: the log of the bestrefs run, and a compressed text file of the ids which are believed to be affected, one reprocessing suggestion per line. The subject line identifies:

1. the project (hst or jwst)
2. use case (test or ops)
3. the old and new context (hst_0052.pmap hst_0053.pmap)
4. the date
5. dispostion (OK, ERRORS, FAIL)
6. count of affected datasets

If the log is longer than 400 lines, only the first and last 200 lines are given.

Worst case, affected_datasets now evaluates 814K datasets in around 3 hours, with run time growing as datasets are added. Best case is around 20 seconds. Run time will get worse as we get more data and improve the affected_datasets tool to include specific table rows. Presumably spending more time on affected_datasets will pay off in eliminated reprocessing.

Sample Mail Subject:

```
FAILED: CRDS hst dev datasets affected hst_0211.pmap --> hst_0212.pmap on 2014-04-23-15:20:27.96 : 0 affected
```

Sample Log:

```
------------------------------------------------------------------------------------------------

CRDS hst dev datasets affected hst_0211.pmap --> hst_0212.pmap on 2014-04-23-15:20:27.96

------------------------------------------------------------------------------------------------

CRDS  : INFO     Mapping differences from 'hst_0211.pmap' --> 'hst_0212.pmap' affect:

 {'acs': ['biasfile', 'drkcfile', 'darkfile']}

CRDS  : INFO     Possibly affected --datasets-since dates determined by 'hst_0211.pmap' --> 'hst_0212.pmap' are:

 {'acs': '2014-03-10 20:37:32'}

CRDS  : ERROR    Failed connecting to CRDS server at CRDS_SERVER_URL = 'https://hst-crds-dev.stsci.edu' :: Required
server connection unavailable.

------------------------------------------------------------------------------------------------

FAILED: CRDS hst dev datasets affected hst_0211.pmap --> hst_0212.pmap on 2014-04-23-15:20:27.96 : 0 affected

------------------------------------------------------------------------------------------------
```

Sample IDs (uncompressed):

```
ibeo59030

ibeo60030

ibeo66rpq

...
```

The IDs include association ids and unassociated exposures. member ids of associated exposures are typically not included.

Majordomo lists

crds_hst_ops_reprocessing@stsci.edu

crds_hst_test_reprocessing@stsci.edu

crds_jwst_ops_reprocessing@stsci.edu

crds_jwst_Test_reprocessing@stsci.edu