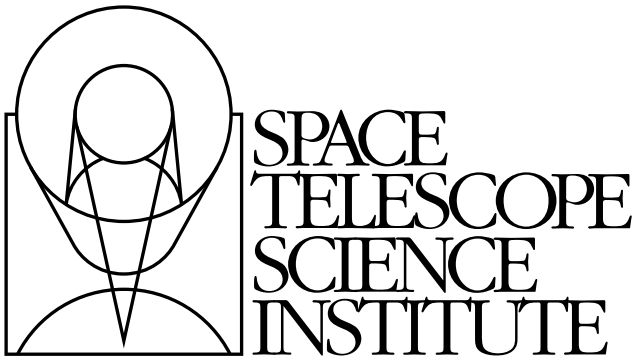

CRDS Server Documentation

Release 1.1

STScI

July 18, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

This guide is intended to introduce the CRDS servers for the purposes of maintenance and emergency backup for Todd. Additional documentation about using CRDS and the servers can be located on the servers under the question mark icon at the top right each page.

Historical Institute contacts for the CRDS servers include:

- Todd Miller: primary CRDS and CRDS server application developer.
- Pey-Lian Lim: intial JWST cloned references and rules, jwst_gentools branch
- Jonathan Eisenhamer table differencing and reprocessing implications
- Patrick Taylor: web proxies, ssl support, and server rc/reboot script coordination
- Thomas Walker: initial VM creation, file systems, and Isilon storage setup.

CHAPTER

TWO

SERVERS

2.1 CRDS pseudo-user and group

The CRDS servers run as the no-login user "crds". The CRDS crds_server script contains the following:

```
% ssh -A -t $1.stsci.edu /usr/bin/sudo /bin/su - crds
```

and is invoked like this:

```
% crds_server plhstcrdsv1
# crds_server <VM-hostname>
```

The ssh first logs into your normal user account on the server VM, then su's you to CRDS. It is not possible to directly login to the crds user.

Server maintainers need to get membership in group crdsoper and "sudo su crds" access on the appropriate server VMs.

Members of DSB share the crdsoper group and use it to modify the CRDS server file delivery directory described below or copy files directly to their ingest directories.

2.2 Virtual Machines and URLs

The CRDS servers exist on virtual machines, run Apache servers and Django via mod_wsgi, are backed by memcached as a memory caching optimization for frequent traffic. Currently there are 6 VMs and servers: (hst, jwst) x (dev, test, ops):

observatory	use	host/vm	direct port	url
hst	django	localhost	8000	http://localhost:8000
hst	dev	dlhstcrdsv1	8001	https://hst-crds-dev.stsci.edu
hst	test	tlhstcrdsv1	8001	https://hst-crds-test.stsci.edu
hst	ops	plhstcrdsv1	8001	https://hst-crds.stsci.edu
jwst	django	localhost	8000	http://localhost:8000
jwst	dev	dljwstcrdsv1	8001	https://jwst-crds-dev.stsci.edu
jwst	test	tljwstcrdsv1	8001	https://jwst-crds-test.stsci.edu
jwst	ops	pljwstcrdsv1	8001	https://jwst-crds.stsci.edu

For debug purposes the servers can be accessed by bypassing the proxy using VM-based URLs such as https://plhstcrdsv1.stsci.edu:8001/. These direct URLs are visible on site only. Only the OPS server URLs will be visible off site.

See CRDS_server/sources/site_config.py to verify server configuration.

2.3 CRDS Client Configuration Scripts

Configuring the client to work with various CRDS servers can be accomplished using scripts define in the CRDS client under trunk/envs:

Script	CRDS_SERVER_URL	CRDS_PATH	Purpose
default	https://crds-serverless-mode.stsci.edu	readonly /grp/crds/cache	Default no env, serverless jwst ops
env-forwarded.csh	https://localhost:8001	$HOME/crds_cache_forwarded	ssh port forwarded private server
env-local.csh	https://localhost:8000	$HOME/crds_cache_local	Django development server
hst-crds-dev.csh	https://hst-crds-dev.stsci.edu	$HOME/crds_cache_dev	Connected hst dev local cache
hst-crds-test.csh	https://hst-crds-test.stsci.edu	$HOME/crds_cache_test	Connected hst test local cache
hst-crds-ops.csh	https://hst-crds.stsci.edu	$HOME/crds_cache_ops	Connected hst ops local cache
jwst-crds-dev.csh	https://jwst-crds-dev.stsci.edu	$HOME/crds_cache_dev	Connected jwst dev local cache
jwst-crds-test.csh	https://jwst-crds-test.stsci.edu	$HOME/crds_cache_test	Connected jwst test local cache
jwst-crds-ops.csh	https://jwst-crds.stsci.edu	$HOME/crds_cache_ops	Connected jwst ops local cache
crds-readonly.csh	https://crds-serverless-mode.stsci.edu	readonly /grp/crds/cache	Disconnected, complete ops cache
hst-crds-readonly.csh	https://hst-crds.stsci.edu	readonly /grp/crds/hst	Connected, complete ops cache
jwst-crds-readonly.csh	https://jwst-crds.stsci.edu	readonly /grp/crds/jwst	Connected, complete ops cache

At present only connected clients can resolve symbolic/date-based contexts other than -operational.

Even without env settings, many tools can guess the appropriate cache and ops server url based on files.

CHAPTER

THREE

MAILING LISTS

The following CRDS mailing lists are defined on behalf of CRDS:

mailing list	moderator	purpose
crds@stsci.edu	mcmaster	comm between INS and CRDS team, also accidental use
crds_team@stsci.edu	mcmaster	comm within CRDS delivery team, DSB, archive, pipelines
crds_datamng@stsci.edu	mcmaster	common destination for affected datasets output
crds-servers@stsci.edu	jmiller	little used, new source address for affected datasets, server details + news
crds_hst_ops_reprocessing@stsci.edu	jmiller	affected datasets output list, hst ops
crds_hst_test_reprocessing@stsci.edu	jmiller	affected datasets output list, hst test
crds_jwst_ops_reprocessing@stsci.edu	jmiller	affected datasets output list, jwst ops
crds_jwst_test_reprocessing@stsci.edu	jmiller	affected datasets output list, jwst test

These are all closed lists. Most critical are crds_team and crds_datamng. The reprocessing lists are intended and/or used to drive automated systems so be careful with traffic on those.

CHAPTER

FOUR

SERVER FILE SYSTEMS

4.1 Cross-Server Shared Home (/home/crds)

The VMs and servers share a common /home/crds directory which has potential as a single point failure. In particular, critical shell rc scripts (.setenv) are shared by all servers and must be updated with extreme care because any error instantly affects all 6 servers.

/home/crds is useful for communicating information between VMs during setup and maintenance.

The RC scripts are version controlled with the server source code in the directory "hosts" under the names dot_setenv and rc_script.

4.1.1 .setenv

The CRDS user runs under /bin/tcsh and executes .setenv for CRDS-server specific initializations. Note that $HOME/.setenv is shared across all CRDS servers and should be modified with extreme caution. The environment variables defined to differentiate the 6 CRDS servers are, for example for JWST DEV:

```
CRDS_PROJECT        jwst
CRDS_USECASE        dev
CRDS_SERVER         dljwstcrdsv1
CRDS                /crds/data1/dljwstcrdsv1
PATH                /crds/data1/dljwstcrdsv1/CRDS_server/host   /crds/data1/dljwstcrdsv1/crds_stacks/
CRDS_STACK          /crds/data1/dljwstcrdsv1/crds_stacks/crds_11
CRDS_AFFECTED...    jmiller@stsci.edu eisenhamer@stsci.edu
CRDS_IFS            /ifs/crds/jwst/dev
CRDS_FILE_CACHE     /ifs/crds/jwst/dev/file_cache
CRDS_SERVER_FILES   /ifs/crds/jwst/dev/server_files
META_PREFIX         /crds/data1/dljwstcrdsv1/crds_stacks/crds_11
```

Additional environment variables, particularly those related to server installation, are defined in ${CRDS}/CRDS_server/env.csh.

META_PREFIX is roughly equivalent to /usr/local, the common value passed to –prefix in ./configure, etc., for building the server Python stack.

4.1.2 .alias

CRDS augments the standard .alias file with these aliases for moving around the file system:

```
#  Source code areas
alias crds       "cd ${CRDS}/CRDS"
alias server     "cd ${CRDS}/CRDS_server"
alias stack      "cd ${CRDS_STACK}"
alias installer "cd ${CRDS}/crds_stacks/installer3/build"

# Server maintenance areas
alias logs       "cd ${CRDS}/server/logs"
alias backups    "cd ${CRDS}/server/db_backups"

# CRDS code area
alias libpython  "cd ${CRDS}/python/lib/python"

# Server working data files areas
alias deliveries "cd ${CRDS_SERVER_FILES}/deliveries"
alias catalogs   "cd ${CRDS_SERVER_FILES}/catalogs"
alias ingest     "cd ${CRDS_SERVER_FILES}/ingest"
alias file_cache "cd ${CRDS_FILE_CACHE}"

# Isilon and VM file systems
alias ifs        "cd ${CRDS_IFS}"
alias data1      "cd ${CRDS}"
```

4.1.3 rc_script

The /home/crds/rc_script is executed to restart the servers, or shut them down, whenever the server is rebooted.

4.2 Server Static File Storage

The CRDS server code and support files (Python stack, logs, monitor_reprocessing dir) are stored on a private VM-unique volume named after the host, e.g. /crds/data1. This serves as the ./configure –prefix directory for a small number of packages not contained in the crds_stack subdirectory. Files within this directory tree are logically executable or in some way secret, sensitive with respect to server security. Most files/subdirs are located in a subdirectory named after the host, e.g. /crds/data1/plhstcrdsv1.

4.2.1 server runtime directory

A number of subdirectories are used to store files related to running Apache, logging, or backups under e.g. /crds/data1/dlhstcrdsv1/server.

conf subdirectory

Apache config files are installed here at e.g. /crds/data1/dlhstcrdsv1/server/conf. Files ssl.conf and httpd.conf.

logs subdirectory

There are a number of Apache logs kept at e.g. /crds/data1/dlhstcrdsv1/server/logs. These logs record requests to Apache and stderr output from Django views not visible to end users.

db_backups subdirectory

The output of the CRDS_server/tools/backup_server script is kept here in dated subdirectories, e.g. /crds/data1/dlhstcrdsv1/server/db_backups/2014-05-30-033327. These contain a backup of the CRDS server database, catalog files, deliveries files, all mappings, the server CRDS cach config directory, and an VM rpm listing. For use with restore_server, these files would need to be copied to differently named locations in $HOME/backups which only record the results of the last backup.

wsgi-scripts subdirectory

The mod_wsgi script which bridges from Apache to Django, crds.wsgi, is kept here, e.g. /crds/data1/dlhstcrdsv1/server/wsgi-scripts. Potentially other django or non-django WSGI scripts would go here as well.

run subdirectory

The running Apache process id is stored here. The id of memcached should be stored here as well but isn't stored.

4.2.2 database directory

Files required to support operations with databases are stored in a top level static file system subdirectory, e.g. /crds/data1/database. These files are secret, effectively mode 700, and maintained manually as part of database setup. They're referred to by site-specific database configurations.

4.2.3 CRDS client source directory

The checkout of the CRDS core library source code installed with the CRDS server is located in the static file tree under the subdirectory CRDS and visited using the alias "crds". e.g. /crds/data1/plhstcrdsv1/CRDS. Typically the server uses the core library and utilities directly, but the server is also responsible for testing the client JSONRPC services.

4.2.4 CRDS_server source directory

The checkout of the CRDS server source code is located in the static file tree under the subdirectory CRDS_server and visited using the alias "server". e.g. /crds/data1/plhstcrdsv1/CRDS_server

sources directory

This directory contains the Django server and application source code.

e.g. /crds/data1/plhstcrdsv1/CRDS_server/sources

- ***sources/configs***
  contains site specific django configuration and database configuration files. The appropriate files are copied to sources/site_config.py and sources/crds_database.py at install time. Those are then imported into more generic configuration files sources/config.py and sources/settings.py. The site specific files are intended to contain the minimal information required to differentiate servers.
- ***sources/urls.py***
  defines most of the site URLs for all applications.

- ***sources/settings.py***
  fairly standard Django settings.py
- ***sources/templates***
  contains web template base classes. many applications also contain a *templates* subdirectory.
- ***sources/static***
  contains most CRDS static files, particularly Javascript and CSS.
- ***sources/interactive***
  is the primary web application for CRDS browsing and file submission.
- ***sources/jsonapi***
  is the JSONRPC application which supports web services in the crds.client api.
- ***sources/jpoll***
  application supports the Javascript logging + done polling system used for long running views, particularly file submissions which can exceed proxy timeouts and run too long to leave a human without info.
- ***sources/locking***
  application for database based locks used by CRDS web logins for exclusive access to an instrument.
- ***sources/fileupload***
  application supports the fancy file submission file upload dialogs for file submissions.
- ***sources/stats***
  application mostly defunct django-level request logging to database, superceded by Apache logging. Some parameter capture not present in current Apache configuration.

host directory

The CRDS_server/host subdirectory is on the PATH. It contains scripts related to cron jobs, affected datasets reprocessing, stack building, server utilities, etc. e.g. /crds/data1/plhstcrdsv1/CRDS_server/host

tools directory

The CRDS_server/tools directory contains more complicated scripts related to server backup, restore, mirroring, consistency checking, server initialization, user and group maintenance, etc. The tools directory is not on the PATH and contains more eclectic scripts developed in an unplanned manner, basically capturing whatever I needed to do repeatedly or had to Google. e.g. /crds/data1/plhstcrdsv1/CRDS_server/tools

servers directory

e.g. /crds/data1/plhstcrdsv1/CRDS_server/servers

This directory contains the Apache and mod_wsgi configuration files which are copied by ./install to their CRDS server installation directories.

4.2.5 crds_stacks directory

e.g. /crds/data1/plhstcrdsv1/crds_stacks

The crds_stacks subdirectory contains mostly stock python stack binaries and source code, supporting third party packages for the server application. The CRDS server Python stack is built from source contained in the installer3 subdirectory. Binaries are output to parallel subdirectories, e.g. crds_11.

An automatic nightly build and reinstall of the stack occurs on the dev and test servers so it's possible to upgrade all the non-ops servers by updating the central installer3 repo at /eng/ssb/crds/installer3.

Independent checkouts of the repo are contained in the stacks file store for each VM. The purpose of individual VMs is to facilitate independent configuration and test of Linux, the Python stack, and the CRDS server on each distinct VM. The OPS servers are configured for manual updates.

4.2.6 monitor_reprocessing directory

Output from the monitor_reprocessing cron job is stored in dated subdirectories here. Also the file old_context.txt which records the last known operational context against which changes are measured. Changed old_context.txt will trigger an affected datasets calculation as will changing the operational context on the web site.

4.3 Server Dynamic File Storage

For operating, the CRDS servers require a certain amount of dynamic storage use for purposes like:

- holding pending archive deliveries (deliveries, catalogs)
- uploading files (uploads, ingest, ingest_ssb)

The server dynamic file storage is located on the Isilon file server at:

/ifs/crds/<obsevatory>/<use>/server_files, e.g. /ifs/crds/hst/ops/server_files.

Since this area is actively written as a consequence of users accessing the web site, it is kept distinct from the code and files required to run the server.

4.3.1 catalogs subdirectory

Files submitted to the archive generate .cat file lists which are stored permanently in the catalogs directory. Any file in CRDS is also stored in the server file cache, so given the .cat file list the delivery can be recreated by regenerating file links in the deliveries directory. The catalogs directory is an internal CRDS server data store which records file lists from past deliveries.

4.3.2 deliveries subdirectory

The deliveries directory is cross-mounted between the CRDS server VM and CRDS-archive-pipeline machines, not necessarily under the same path name.

Files submitted to the archive are placed in the CRDS delivery directory along with a numbered catalog file which lists the submitted files one per line. Unlike more CRDS directories, the delivery directory is cross-mounted to pipeline machines which handle archiving. As part of the protocol with the CRDS archiving pipeline, the catalog file is renamed to indicate processing status. When the catalog is finally deleted, CRDS assumes that archiving is successful. See crds.server.interactive.models for more info on the delivery naming protocol. Note that files in the delivery directory are linked to the same inode as the CRDS file cache copy of the file, or, in the case of the .cat delivery file lists, to the permanent copy in the catalogs directory. For references, linking avoids substantial I/O overheads associated with multi-gigabyte JWST references. For catalogs, linked or not, like named file lists should have the same contents in catalogs and deliveries.

4.3.3 uploads subdirectory

The uploads directory is the default Django file upload directory for simple file uploads.

4.3.4 ingest subdirectory

The ingest directory tree contains per-submitter subdirectories which are written to by the Django-file-upload muli-file upload application used on file submission pages. The user's guide gives instructions enabling submitters to copy files directly into their per-user subdirectories as an upload bypass for telecommuters. (This is a work around for the situation in which a VPN user winds up transparently downloading and then explicitly uploading references submitted via the web site; instead, a submitter places the file directly into their own ingest directory keeping the file onsite, then proceeds with the submission on the web server normally.)

4.3.5 ingest_ssb subdirectory

The ingest_ssb directory tree is the historical generation and/or drop-off point for the files generated by the jwst_gentools. Ingested files are then submitted to the web site. The server does not directly access this directory, it shares space with it.

4.4 Server File Private Cache

The Isilon CRDS cache storage (i.e. CRDS_PATH for servers) is located similarly to dynamic file storage:

e.g. /ifs/crds/jwst/test/file_cache/{config,mappings,references}/{hst,jwst}

Each CRDS server (test or ops) has a full copy (~2T allocation) of all operational and historical (CRDS-only) reference files and rules. The dev servers have a smaller allocation which is generally linked to /grp/crds (synced from ops servers) rather than internally stored.

The server file cache config area is generally updated transparently by running cronjobs. The server file_cache and delivery areas are updated as a result of file submissions and archive activity. Once global Isilon archive storage becomes available, cache space can be reclaimed by symlinking the CRDS cache path to the global storage rather than maintaining an internal copy; there should be a lag of a couple weeks to a month between submission and reclamation during which the potentially transient file is fully stored in the CRDS server. Because the CRDS server caches also contain unconfirmed and unarchived files, they are currently read protected from anyone except crds.crdsoper.

See the User's manual in the ? on the web sites for more info on the CRDS cache.

CHAPTER

FIVE

CRON JOBS

Use shell command:

```
% crontab -l
```

to dump the current crontab and observe the jobs. Cronjobs currently produce .log files in the CRDS_server directory.

To change the cronjobs modify ${CRDS}/CRDS_server/host/crontab and then do:

```
% crontab ${CRDS}/CRDS_server/host/crontab
```

Note that systems on the same subversion branch on which a crontab is modified and committed will automatically pick up and use the new crontab during the nightly cron job.

See "man cron" or Google for more info on maintaining the cron table and crontab syntax.

5.1 nightly.cron.job

CRDS_server/hosts/nifghtl directory and executes every night at 3:05 am. The dev and test versions of the nightly cron fully rebuild and reinstall the CRDS servers, with the exceptions of database secret setup, cron jobs, and .setenv rc_script scripts. The nightly cronjob on all servers captures diagnostic information about the server, including server configuration, disk quotas and usage, subversion status for detecting uncommitted changes and observing branch and revision, and cache consistency and orphan file checking. All of the servers currently update subversion although the OPS (and often TEST) servers are typically on a static branch. The dev and test servers also restart. Output from the nightly cron is sent to the MAILTO variable defined in the CRDS_server/host/crontab file, currently jmiller@stsci.edu.

5.2 monitor_reprocessing

Every 5 minutes CRDS_server/host/monitor_reprocessing looks for changes in the CRDS operational context and does an "affected datasets" context-to-context bestrefs comparison when the context changes. This generates an e-mail to the $CRDS_AFFECTED_DATASETS_RECIPIENTS addresses set up by the .setenv file. bestrefs can require from 20 seconds to 4-8 hours depending on the number of datasets potentially affected as determined by file differences.

5.3 clear_expired_locks

Somewhat dubious, this falls into the category of periodic server maintenance, removing expired instrument locking records from the server locking database. Every 5 minutes. Datatbase locks are considered expired when the current time exceeds the start time of the lock plus the lock's duration; since this is an asynchronous event, the expired lock

records sits around in the database until scrubbed out. In theory the expired locks are replaceable anyway but this routine makes sure they're not sitting around in the database causing confusion. This does not produce e-mail.

5.4 sync_ops_to_grp

Every 10 minutes *sync_ops_to_grp* runs crds.sync to publish the crds ops server to the **/grp/crds/cache** global readonly Central Store file cache CRDS currently uses as default for OPUS 2014.3. This does not produce e-mail.

CHAPTER

SIX

MAINTENANCE COMMANDS

Maintenance commands are typically run from the root of the CRDS_server checkout. Changing to the CRDS_server source directory can be done like this:

```
% server # cd to the CRDS_server source code checkout
% pwd
/crds/data1/dljwstcrdsv1/CRDS_server
```

From here on, we'll assume commands are executed from this directory.

The default Python environment does not include the CRDS server packages directory. Additional environment variables required to run the server and some scripts are sourced like this:

```
% source env.csh
```

6.1 Installing the Server Application

Running the *./install* script will perform many actions including regenerating the environment definition script *env.csh*. Primarily *./install* installs the *crds* (core + client) and *crds.server* packages into a server specific python directory which is added to PYTHONPATH automatically in *env.csh*. In addition ./install instantiates some Apache configuration file templates and copies them to the appropriate installation directories.

The install script is typically run like this:

```
% ./install [hst|jwst]  [django|dev|test|prod]  |& tee install.<observatory>.<use>.err
```

For example:

```
% ./install hst dev  |& tee install.hst.dev.err
```

Running ./install explicitly is required to generate *env.csh* for the first time. Afterward, env.csh essentially knows this server is for "hst dev".

6.2 CRDS Catalog Initialization

Historically the CRDS server catalogs were initialized many times from existing CDBS and JWST references and the initial CRDS rules set. *./init* is rarely used anymore but may still be useful for setting up a Django local test environment, dubbed the "django" usecase.

For the most part the ./init script is tasked with installing the server's initial copy of CRDS rules and initializing the CRDS file catalog (the crds.server.interactive.models Django database with 19000 CDBS references and CRDS rules):

```
% server   # alias to cd to server source directory
% ./init [hst|jwst] [django|dev|test|ops]
<enter password for test user>
```

NOTE: At this stage *./init* should not be run on the OPS servers. For VM-based servers it has effectively been superceded by *tools/restore_server* and *tools/mirror_server*.

6.3 Starting and Stopping the Server

The CRDS server can be started and stopped like this:

```
% ./run
% ./stop
```

The *./run* script starts Apache (many httpd processes) and memcached after which the CRDS server should definitely be abvailable on it's private port (typically 8001). The web proxy is provided by an independent system which is rarely-if-ever unavailable, but which has historically had a random lag of about 1 minute to (by appearances) connect with the just started Apache.

6.4 Updating and Restarting

Performing a server update generally revolves around stopping the server, changing and reinstalling the Django application, and restarting the server. This is encapsulated in the *./rerun* script:

```
% ./rerun
```

This works by sequentially invoking other more basic scripts: ./stop, ./install, ./run.

./rerun produces a log file of the voluminous install output as install.<observatory>.<usecase>.err. If things aren't working coherently, check the install...err file to verify that no setup functions failed, as might happen for a Python syntax error or database schema change.

NOTE: rerunning the server is an integral part of taking the sever offline and switching to the hidden backup port. Consequently, activities such as running tests and mirroring should also be viewed as reinstalling the Django application. The reinstall is innocuous because any differences in application source code should be very tightly controlled, related to switching ports only. However, it's still a significant hidden side effect to be aware of because it has obvious implications when performed on an dirty code base.

6.5 Running Server Tests

The CRDS server unit tests (**NOT ADVISABLE FOR OPS**) can be run like this:

```
% ./runtests
```

additional parameters can be passed to runtests, for example to select specific tests:

```
% ./runtests interactive.tests.Hst.test_index
```

Runtests should not be executed on operational or in-test servers because it has side effects which interfere with server operation. Runtests has been modified to switch to a backup port during execution, but the version of code necessary will only be deployed with OPUS 2014.3 so it is not yet in operations.

NOTE: without special arrangements, server self-tests should not be run on the operational servers. Self-tests are normally run on the dev and test servers during the nightly cron job at 3 am.

It should be noted that the server unit tests typically do run on the dev and test servers in the nightly cronjob, generally making them available without waiting on the following day.

The server self-tests exercise most but not all of the Django interactive view code, JSONRPC code, and basic database interface to DADSOPS. Although the interactive (web view) self tests run in a Django test database, the JSONRPC tests simply ivoke the CRDS client routines to call to the server and verify results. Hence, the JSONRPC code is effectively tested against a live server, exercising it just like a normal user. In addition, the Django caching interface is not mocked during testing, so memcached effects impact the live server. Consequently, for running tests on dev, test, or ops servers, runtests moves the server to the "backup port" where it normally hides during server restoration or mirroring. Self-tests are typically run like this:

6.6 Django Management Commands

Django has a manage.py module which is frequently referenced for server maintenance activities. In CRDS this is wrapped as:

```
% ./manage <additional parameters to manage.py>
```

6.7 Command Line Server Debug

An Ipython shell which runs in a context similar to the CRDS server can be started like this:

```
% ./manage shell
In [1]:
```

This shell can be useful for debugging and/or maintaining Django models, view code, JSONRPC routines, or the database interface to the DADSOPS dataset catalog database (HST).

This shell executes in the same directory/context as the CRDS server, so it provides the same interactive environment in which server Django code normally executes. Consequently server modules and packages tend to import and function normally for interactive debug; this happens in a shell processs, not an Apache process, so the principle coupling to a running server would be the database and file system... and potentially memcached.

6.8 CRDS Catalog Database SQL Commands

The CRDS reference and rules catalog is implemented as a Django model in crds.server.interactive.models. Typically it is accessed by using the models module, classes, and functions. Nevertheless, the Django models can be accessed directly with SQL like this:

```
% ./manage dbshell  # to open a SQL prompt to the CRDS server database
...
mysql> ... SQL commands ...
```

The server unit tests are ponderous. Eventually you may <*control-c*> and leave behind a junk test database which blocks subsequent testing. That can generally cleaned up, with **caution**, as follows for e.g. hst dev:

```
% ./manage dbshell
mysql> drop database test_crds_hst_dev;
```

NOTE: the CRDS Catalog is in a Django database which is distinct from the DADSOPS dataset catalog that CRDS uses to find matching parameters and dataset ids.

6.9 Nightly Backup

All 6 servers run a nightly backup job at 3 am EST. The backup dumps the Django database and attempts to capture transient or unique information in the file system. The backups make a full copy of all CRDS rules. The backups do not contain any references, and in particular, no transient references in the process of submission or confirmation. Nevertheless, the backups are extremely useful and appear to be capable of restoring "yesterday's quiescent server".

Making a backup is done as follows:

```
% tools/backup_server
```

backup_server results in the generation of backup files which are placed in **${CRDS}/server/db_backups** in a dated subdirectory with dated names, and also globally in ${HOME}/backups with generic names. Both locations should be considered secret and hidden using file permissions. Dated backups are persistent, the backups in ${HOME}/backups are overwritten every time backup_server is run. There are unique files for each server. The files in ${CRDS}/server/db_backups are only visible on that VM.

6.10 Restoring Nightly Backups

A relatively recent addition is the tools/restore_server script. It is quite simple to restore the nightly backup of a server:

```
% tools/restore_server
```

Conceptually, restore server reloads the server database and restores the delivery directories and catalogs, and removes any reference or rules files orphaned by the database restoration, those added to the cache since the backup was made.

As a matter of implementation, server restoration is handled by mirroring a server to itself.

During the process of restoration, the server is moved to a hidden backup port and will be seen as temporarily unavailble through the proxy.

restore_server utilizes the backup files in ${HOME}/backups, nominally the ones from the last time backup_server was executed. There is currently no automatic process for appropriately copying the dated backup files from ${CRDS}/server/db_backups to ${HOME}/backups so they can be used in server mirroring or restoration.

IMPORTANT: restore_server should only be used on the OPS server under duress. Prior to restoring the OPS server, review the restore_server / mirror_server and attempt to mirror the OPS server down to a DEV server, then test the mirrored DEV server both interactively and with runtests.

6.11 Server Mirroring

The term *server mirroring* is given to the process of transferring the server database and file system state from one VM and server to another, effectively making the destination server a copy of the source server.

Typical mirroring flows would be to copy the HST OPS server down to the TEST or DEV server, or TEST down to DEV.

Server mirroring leverages (nightly or dynamic) server backups by restoring them to different servers. Afterward, the sync tool is run to synchronize the destination cache with the source server. Subsequently, the tools/orphan_files script is run to verify destination server file system consistency with the destination server file catalog.

mirror_server does not safeguard against it, but it is almost certainly an error to run mirror_server on an OPS VM, which in all likelihood replaces OPS state with something inferior. There is one exception: *restore_server* will mirror the OPS server to itself by running *mirror_server* internally in order to revert OPS to its state at the time of the nightly backup.

For example, to copy the test server (hst-crds-test, tlhstcrdsv1) down to the dev server (hst-crds-dev, dlhstcrdsv1), perform these steps.

First, optionally, on the source server:

```
# login tlhstcrdsv1
% server
% tools/backup_server
```

That puts required backup files in global (cross-server) ${HOME}/backups. If this steps is omitted, the files in ${HOME}/backups should correspond to the server state at the time of the last backup, nominally 3 am. If you're trying to mirror a change on the test server that you just made, then immediately backing up the test server is required so that the change is recorded in the current backup.

Second, on the destination server:

```
# login dlhstcrdsv1
% server
% tools/mirror_server hst test https://hst-crds-test.stsci.edu |& tee mirror_server.hst.test.err
```

where the parameters to mirror_server specify the *source* server and the destination is implicitly the server of the current login.

Server mirroring requires the source server to be online and available. The destination server is moved to a backup port so that it is unavailable while it transitions through various inconsistent states.

CHAPTER

SEVEN

DELIVERY TROUBLESHOOTING

This section discusses possible operational failure modes and how to handle them. There are some comaratively simple problems which may be addressable on an emergency basis. As a general rule, for seemingly complex or uncertain procedures, first mirror the OPS server to the DEV server, then perfom the procedure on the DEV server, then apply the proven procedure to the OPS server. For improved certainty, switch the DEV server to the OPS server source code branches (CRDS and CRDS_server), rerun, and then perform the procedure on the DEV server.

7.1 Remedy by Backup

For some failure modes it may be desirable to restore the server to the nightly backup for the previous day. See *restore_server* above.

NOTE: requires server database and file store changes, restart.

This approach might be particularly effective for temporarily bypassing failed deliveries by one instrument so that others can proceed, and also for cleaning up new or failed CRDS rules which are known to be non-viable. If failed CRDS rules have already been transferred to the archive, either removing them from the archive must be coordinated with DSB and the CRDS Archiving Pipeline, or restore_server should not be performed and the files should be Marked Bad in CRDS instead. See the CRDS user's guide (on the server) for information about marking files as bad.

7.2 Rmap or Context Fix Required

Potentially a best references assignment error could be detected which requires a rules fix.

NOTE: should be possible without OPS server changes.

The procedure for fixing rules should basically be:

0. Mirror OPS to DEV server and work using the DEV server.
1. Sync or download a copy of the rmap file requiring changes.
2. Correct the rules and test locally using elevated verbosity. –verbose or –verbosity=100 or something in between.
3. Upload the modified rmap using Submit Mappings and check "Generate Context" to create new instrument and pipeline mappings which include the new context.
4. DEV servers do not archive and rules are immediately sync'able and useable. Sync to a local cache and test.
5. When satisfied that the DEV server is working, repeat for the OPS server. Very possibly the original fixed copy of the .rmap is directly submissible to OPS.

6. When the OPS systems have successfully archived the new rules, test them by syncing and running bestrefs. The default readonly cache at /grp/crds/cache should sync within 15 minutes of archiving.
7. Inform crds_team@stsci.edu that you think the new rules are working and should either receive a second opinion or be made operational by the pipeline, basically Richard Spencer performing Set Context on the web site.

Context fixes (imap's and pmap's) need to be performed manually, typically without automatic renaming. New Context files are still submitted using Submit Mappings, but without file renaming or context generation.

7.3 Improper Reference File Constraint

Valid reference files may be rejected due to overly stringent or incorrect matching parameter constraints.

NOTE: requires OPS server file updates, reinstall, and restsart.

The synposis of the fix is to modify the appropriate .tpn and/or _ld.tpn file in crds.hst.tpns and update and restart the server.

It's possible that the reference file constraints defined in the CRDS observatory packages will be overly stringent causing the submission of a valid file to fail. For HST, reference constraints are defined in the crds/hst/tpns directory and define two phases of reference file symbolism. The first phase, defined by .tpn files for each instrument-specific type, defines reference parameters as they appear in the reference file. The second phase, defined by _ld.tpn files, define reference parameters as expanded in rmaps by rules in crds.hst.substitutions. In this scenario, errors or missing values in the .tpn's need to be fixed.

7.4 Improper Reference Parameter Expansion

Valid reference files may be inserted into their corresponding .rmap incorrectly, most probably identified by certify warnings about new match tuples in the updated .rmap.

NOTE: requires OPS server file updates, reinstall, and restsart.

The synposis for the fix is to modify substutions.dat in crds.hst, reinstall the server, and restart.

For HST, reference file matching parameters define where the reference is inserted into .rmaps. During the reference insertion process, reference file parameters are expanded using context-sensitive expansion rules defined in crds/hst/substitutions.dat. Deficiencies in those rules will result in references being added to the wrong rmap matching paths. The short term fix would be to modify substitions.dat, manually test the rmap update proces, the resulting rmap, and finally adjusted best references.

7.5 Table Row Change Warnings

Submission of a new table reference file may result in certify warnings due to comparison with the old version of the table and deletion of rows.

NOTE: make it clear warnings are approximate, tripwires, then verify file differences and confirm or cancel.

It should be noted that the warnings are approximate and advisory, not definitive. With that in mind, verify with the submitters and/or reference developers that the noted differences are not a problem, then proceed with confirmation or rejection. Row modifications may be perceived by certify as deletions and additions rather than as replacements.