**WORKFLOW**

This note describes a draft plan for pipeline process management. It is based on a simple model where each data product has some tabulated information describing how to make it. The assumption is that each product has a single specific program (usually a script) that builds it. There is a set of three items, a data product, a gate, and an action task that are tied together in this approach. Further, at least for now, each product has a single ordinate variable, i.e. is like MDI dataseries which have a single "time" axis. Since the main goal of the pipeline process management is to automate processing as new data arrives, the use of time or a function of time as a primary organizing concept seems OK.

The model has 5 basic components:

1. "gate" A gate is to be thought of as an order desk or order window or gate in a fence opening onto a subset of a large span of data. Perhaps "window" would have been a better term, but it is well enough overloaded already. A gate is the access point to tell a user or program about the availability of the gated dataseries for a desired range of "time". Time can be real time, or FSN, or Carrington time, or other single quantity which increases monotonically with the addition of new data. In cases where multiple quantities are used to define the product, there will still be a primary axis which will be called "time" here. A gate can be accessed with a ticket. A ticket contains information about the span of desired time and an action. The action can be to query for the existence of data covering the desired span or to cause the data to be created for that span. The gate contains links to tasks that know how to do the query or to start the product computation. A gate may have multiple tickets in its "new" and "active" queues.

2. "ticket" A ticket is a message that is passed from a task to a gate and returned by the gate to the calling task when the action is complete. A ticket is simply a message. All tickets are originated by a task and returned to that task when their requested action is satisfied.

3. "task" A task is program to either query for the span of data available a given product or a task to create a product for a given span of time. The create task ("action task") is usually a script that when executed will itself start additional scripts to run via qsub in a drms_session to generate some data product. A task will usually have prerequisites that must be met before the action task can start. These are the existence of source data for a span of time. The action task description contains a list of preconditions and instructions for building tickets to ensure that the required data will be available. Prior to the start of a task, tickets will be issued to gates corresponding to each source data product needed to generate the target data. When those pre-condition tickets are returned, then the computation part of the action task will be started.

4. "taskmanager" A taskmanager is a script that is called by the gatekeeper when an action task needs to be started. The task manager is a script that actually examines the prerequisite list and issues and waits for tickets for the needed input data. An instance of a taskmanager script is started for each span of a data product that will be computed. That taskmanager instance will survive only until the product is ready. A taskmanager is initiated by the gatekeeper.

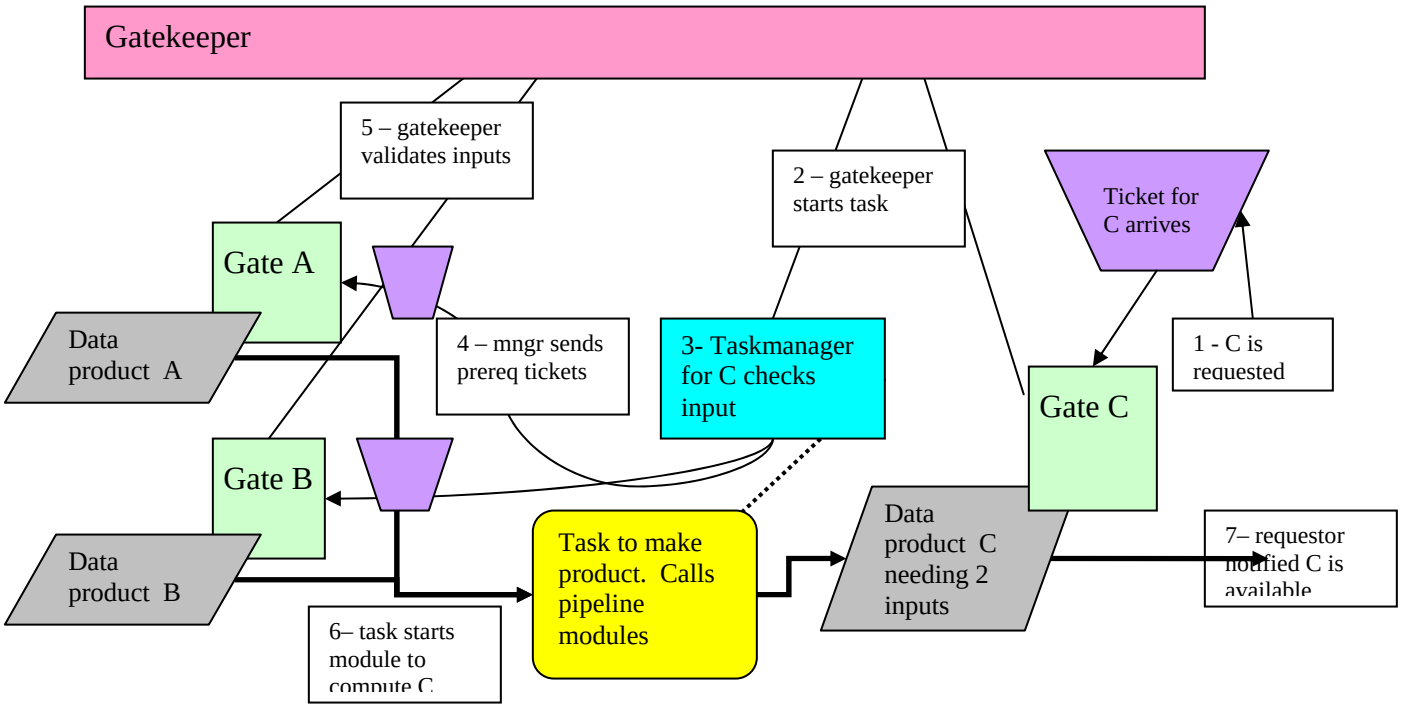
5. "gatekeeper" there is a single gatekeeper deamon process running. The gatekeeper regularly polls all gates for new tickets. When tickets are found

the gatekeeper examines each ticket and disposes of it either by running the query task (a.k.a. status task) and returning the ticket or by starting the action task. When the action task is done it will update the status value in the ticket and move it to the task's done queue. Then sometime later the gatekeeper will find that the ticket is satisfied and in the done queue and will return it to its parent task.

The combination of these five component parts is a system that calls itself recursively to satisfy the input conditions enabling a final product to be computed. These five components are each simple in concept and in most cases simple in implementation. The complexity is in the static data contained in gate descriptions and task descriptions. It is possible that there is a single taskmanager code that can be stated for all action tasks. There is a single gatekeeper. All tickets have the same structure and all gates and tasks have the same basic elements.

The global status information can be obtained by scanning all gates and noting the current ticket activity. Processing errors can be indicated on return tickets so appropriate action can be taken. The structure of the system implies a "pull" type approach to data workflow, so in the cases where a "push" is more appropriate such as for quicklook processing, a parent request task can be left running to periodically issue tickets and cause the next expected increment of data to be pulled through.

In the description below, the five main components are described in general terms of their attributes first. Then a draft-implementation is described using files and directories. The system could be implemented in a database as well, or in multiple ways between the simplicity of files and directories and database tables. There are some advantages to the file based approach in terms of simplicity and ease of monitoring. It is easy to visualize a tool that would scan the gates (directories) and display the current status to a pipeline flow manager web page.




**A simple diagram shows the relationships of these 5 colored components.**

Examples of products where multiple prime keys may be needed, i.e. those
For which the ticket system described here is a problem.
In no particular order:

1. Active region disk passages. Expect both ARID and time keywords.
For each ARID there is a time sequence to be built up as it transits the disk.
There are few enough ARIDs that for each set of processing the ARID could be
considered a constant but there will be a big gain in doing all the on-disk
ARIDs in parallel from the one set of input data, which is a time series of
images. So there is a set of ARID to update for a range of times.

2. Spherical harmonic tiles. There are a set of tiles to be updated
From time series of spherical harmonic amplitudes. Best to update many output
series from a single input time series. So the tiles could be considered a list
of targets to update for a range of times.

3. tracked regions for local helioseismology. In both the T-D and ring cases
there will be a list of Carrington coord specified tiles to be tracked for a
span of times. Each tile that has data from a given time should be updated in a
single pass through the time range. So a list of tiles can be generated for a
range of time.

One of the goals of the prototype exercise is to evaluate the actual
requirements for processing this type of data product.

**Support Programs and test implementation**

In a test implementation several scripts have been defined to encapsulate details. These are:

maketask.csh – takes a set of keyword=value arguments and generates a task as a set of directories and files. In addition to calling maketask a task with preconditions needs to have a directory named for each prerequisite gate created in the precondition directory created for that task by maketask.csh.

makegate.csh – takes a set of keyword=value arguments and generates a gate as a set of directories and files.

In the test implementation both gates and tasks have attributes in files named for the attribute and containing a single line containing the value part of the pair.

maketicket.csh – makes a ticket from a set of keyword=value arguments. The ticket is implemented as a single file containing one line per keyword=value pair.

wait_ticket.csh – takes single argument of ticketid, waits until the actions requested by the ticket are complete and returns the ticket contents and path to the task instance used for the processing. This program is not used internally, but allows users to manage calls to the pipeline manager.

GetNextID – C program which makes a unique identifier based on a seed, the date, and a sequence number unique on each day. Is used to name tickets and task instances.

init_pipeline.csh – takes no arguments but creates the basic directory structure needed for all other components. Used only to reset everything.

add_<product> scripts are scripts that create gate:task sets by calling maketask, makegate, and maketicket for initialization tidkets. The add_XXX scripts made so far also contain code which removes any prior gates and tasks by the same names since they are used for developing the system. They should be run with caution. In many cases they are out of date and the current setup should be compared in detail to the add script before use.

watchgates.csh,
checkgates.csh,
watchtasks.csh,
checktasks.csh,
A set of simple-minded status viewers. Prints list of active tickets for each gate, or list of instances for each task. For task instances also show which queue they are in and the status of any pending tickets. The “watch” versions repeat after a 30 second sleep while the “check” versions make only one pass through the selected list of gates or tasks. A single argument is allowed which is a pattern to match to pick the gates or tasks to monitor. For example the command:

```
watchgates.csh LOS
```

will show only gates with LOS in their names. The “watch” versions terminate monitoring when they are killed by the user with e.g. a CTRL-C. All of these must wait until the gatekeeper is sleeping to avoid looking at directories or files that are changing or moving. They issue a ‘.’ each second while waiting. There are no locks so these scripts occasionally die when a ticket or directory

go away after being detected but before being used to get status. So if the "watch" scripts die, just restart them. (it would be nice to have file locks but that is not practical in the current implementation).

All of these scripts are found at …/cvs/JSOC/proj/workflow. They all expect the environment variables WORKFLOW_ROOT to be set to that or another location where the scripts exist. The scripts also expect an environment variable WORKFLOW_DATA to be set to the path to the working directory containing the gates and tasks.

The working scripts for tasks, status tasks and action tasks are kept in the scripts sub-directory of WORKFLOW_ROOT.

Things to add to the test:

The coverage map is not implemented and I believe will be important to let the system have good performance since it could be easy to maintain but will be expensive to create from scratch. Should tell which "slots" of each gated series are filled with data and which are permanently missing. Such a tool could be useful in redoing products – all that would be needed is to delete the coverage map entries for the times that need to be recomputed. Then run a tool that would generate tickets for each gap. At present action==4 is handled by using the show_coverage command to identify gaps that need data to be computed.

Error handling is not robust. So far detected errors are propagated into ticket STATUS keywords and task instance state keywords. If each task puts intermediate log information in the task instance directory – which is available for this purpose so long as the reserved names are not stomped on – then one could inspect the instances in error state and figure out what failed. During testing the difficult bugs to find were when an actiontask script failed without leaving a trail. Then the watch tool showed tickets and task instances just sitting in an operating state but no clue as to why. Return values for tasks are captured into state and STATUS keywords. When a failed ticket is detected that information is copied into the $WORKFLOW_DATA/FAILED_TASKS file. This file should be watched for failures and contains the full path to the failed instance directory.

Stopping and Restarting: In principle the gatekeeper can be stopped and restarted at each transit around its main loop. Each loop it looks for a file named "Keep_running" and terminates if that file is absent. It touches a file named GATEKEEPERBUSY while it is in sensitive areas – changing tickets. This is used to block access to the gate directories while they are changing so the watch scripts will not get inconsistent states. Stopping and restarting the gatekeeper works OK since only the gatekeeper changes the contents of tickets and only the taskmanager starts actiontasks. However a complete kill of running action tasks is harder to accomplish and recover from. The way to do it is to remove top level tickets with "cancel_ticket.csh" which tries to remove the tickets, sub_tickets, task manager instances, etc. Tasks processing data in the cluster via qsub may be stopped with qdel or let run to completion. Then resubmit the top-level tickets. To help do this a history is kept in xxxx.history for created tickets.

System has been tested for subrange generation and for preconditions. One thing that should probably be changed in a complete implementation would be to submit preconditions first and range parts second – or at least to have that option since it may be more efficient to make a larger set of preconditions vs. doing them in segments of maxrange. Not really sure about this though. Now the

taskmanager first breaks the ticket span into subtickets if it is larger than the maxrange value. When all of those are done, it exits. When the range is less than maxrange, it checks for preconditions and issues tickets for each precondition then when all are satisfied, does the primary action task. I.e. the taskmanager causes itself to be called recursively. This order is probably correct for low level products such as hmi.X_45s since it lets each chunk proceed when preconditions have been met. For some processing steps the other way would be better since some programs are more efficient in larger chunks, e.g. making hmi.cosmic_rays for a full day in one call instead of a bunch of 24 minute segments.

Preconditions are implemented as new tickets. The precondition list is implemented as a directory containing directories named for the prerequisite gate names. Inside those directories MAY be a file called ‘prepare_ticket’ which will be ‘sourced’ just before maketicket is called. Just before that each of the args for maketicket are set in variables. Thus those variables may be modified if needed before starting the precondition ticket. In the example set this was used to force the precondition ticket to ask for ACTION=5 vs. the default ACTION=4, the difference being that 5 implies always call the action task of the precondition while 4 will first check to see if the precondition data is present. For the test to be run over and over I needed to have it do the tasks each time – over and over. In real cases the ACTION=4 is clearly the choice. 5 would be used only when needing to force re-computation of a product. Action=5 is also used in the clock-repeat pair of gates and tasks. When computing a segment of data for the first time, ACTION=4 and ACTION=5 are almost the same but ACTION=4 has a bit more overhead.

**TICKET**

A ticket (or action request) is a message containing a request for information to be provided or action to be taken for a particular data product. A request for a specific product is constructed as a ticket which is delivered to a "gate" or "order window". There is a gate for each data product. The gate has information about how to query for availability of some of its data and about the tasks needed to extend the range of data. A ticket does not return data, only information about the availability of the data.

A ticket is a message that can be registered with a gate by a task.
A ticket is used to ask for status from a gate or to wait until a requested state is available.
A ticket is defined by a prototype and is created/instantiated by a task.
A ticket defines a desired range of (at present) a single axis or ordinate variable. A fixed set of "axis" types are available including time, serial number, and Carrington time for now.

Attributes:

```
# set by the maketicket call by an originating task instance:
  GATE
      Name of gate that this ticket is sent to or is returned from
  SPECIAL
      Special requirements for this ticket, will be a list of prime key values
      for prime keys other then the controlling KEY.  Value part should be
      grouped into 'words' that can be evaluated as KEY=VALUE pairs
      by csh or other interpreter.
  ACTION
      What is request for gate
            1.  Tell the full pass range of values
            2.  Is gate open to the "want" range of values
            3.  Wait until gate open for all of want range
            4.  Push the gate to open to the requested range if needed
            5.  Force gate to recomputed to requested range
            6.  Report gaps between WANTLOW and WANTHIGH, method
                of report is implementation dependent. Permanent
                gaps are not reported.
  WANTLOW
  WANTHIGH
      Values of gate's control keyword that the ticket is asking about or
      requesting

 # set by task sending ticket to a gate
  TICKET
      This ticket's name, same as ticket filename in test implementation.
  TASKID
      Task ID of task instance that originated the ticket
  EXPIRES
      Time at which ticket times out, presently set as 3 from NOW by
      maketicket.csh

# set by gate when returning ticket to task:
  STATUS
      Response to action
            0:  Gate is OK to pass, want range is "open"
            1:  Gate is closed to all or part of want range
```

```
            2:  Ticket is new, no valid status yet
            3:  Ticket is being processed by gate
            4:  Time out - ticket is too old
            5:  Error, request makes no sense or resulted in a processing error
            6:  Cancel – ticket has been cancelled.
  GATELOW
  GATEHIGH
        For STATUS 0:
          Response low and high values.
            Full range for action 1.
            Overlap range for actions 2, 3, 4
        For STATUS 1:
          Response low and high values.
            Overlap range for actions 2, 3, 4
            missing values for action 1 - only happens if
                      product does not have any data.
        For STATUS > 1, these are not defined.

#########################################################################

Notes about ACTION:

Action = 3 presently tests only for high and low values not for completeness
within the range.  this is a flaw.  Test should be as for action = 4

Action = 4 uses show_coverage and thus fails for non-slotted series.  At present
only hmi.cosmic_rays causes difficulties and led to adding the optional
“coverage_args” argument in the gate description.

Implementation:

A ticket is implemented as a file containing keyword=value pairs, one per line.
Everything after the “=” is part of the value (except leading or trailing
blanks).

Tickets are created by the task manager using prototype information in
a task description.  The task manager creates a ticket file in a gate's
new_ticket directory by calling maketicket.csh.  When all work is done on the
ticket, the gate manager (gatekeeper) moves the ticket into the task's ticket
return directory (“done”).

A ticket has a name built from the gate name and sequence number computed
from the CURRENT_TICKET number in the gate directory.

If the gap list is requested (action 6) the list is provided as a file in
the ticket return directory, name of file is ticket TASKID with suffix ".gaps"

TASKID contains the generating task instance taskid.
e.g.:  TASKID = <task_name>-<current_task>
TASKID is same as <ticketid> and is used as a sub-directory name in the tasks
active, done, archive/ok and archive/failed directories.

The SPECIAL attribute is a string containing specific instructions for this
ticket for the target gate.  It will usually contain lists of associated prime
key values for the target series.  Examples might be active region numbers as
targets for the range of times given in the principal axis.
```

Difference between ACTION 3, 4 and 5 is that 3 will simply wait until magic happens and the product becomes available, 4 will check to see if the ‘gate is already open’ to the wanted range and only execute the update task if the product is not already available.  Action 5 will not do this initial check so the target data will be recomputed even if already present.

ACTION 3 can be either wait for wantlow and wanthigh in range or for full coverage from wantlow to just less than wanthigh depending on the code in the statustask.  Should be appropriate for the product.  At present the test is done by the gatekeeper and only checks for wantlow to wanthigh being contained in the range low to high.

While the “product” information can be retrieved from the gate which is given in the ticket, the path to the gate may be implementation dependent so it would be a good idea to include product in the ticket keywords.  Several action tasks have been written to be reusable and need this information.

**GATE**

A gate is an interface to a data product, it describes availability of data. A Gate has a range of current validity, low to high of some quantity usually expressed as a pair of times but any numerical value will do. Times expressed as DRMS time strings. Each gate is essentially the keeper of the available data times for a single product.

A gate has the following attributes.

NAME
: Name of gate, should indicate associated dataseries and quantity. The gate name may not contain a dash, “-“.

GATESTATUS
: Values are “ACTIVE”, “HOLD”. A status of HOLD stops all ticket processing for the gate.

PRODUCT
: DRMS seriesname of the product which is controlled by the gate. There can be multiple gates for a given PRODUCT in the case where the PRODUCT series has multiple prime-keys that specify specific cases that are managed independently.

LOW
HIGH
: Values of the limits of the open region of the gate.

TYPE
: Kind of "axis" for the gate control values. e.g. "time", "SN"

KEY
: Prime key name for the main “axis” described by LOW, HIGH, and TYPE.

LASTUPDATE
: Time of last update of the gate

NEXTUPDATE
: Time of next expected update of the gate range

UPDATEDELTA
: Time increment between scheduled or expected updates.

STATUSBUSY
: Flag indicating STATUSCOMMAND is running, locks out use of the gate.

STATUSCOMMAND
: Command to run that will update the gate status. Command takes one argument which is gate name. It updates LOW, HIGH, and LASTUPDATE values. If LOW is not MISSING it is not updated and the current HIGH limit if the data is discovered. I.e. LOW is only updated the first time unless something sets it to MISSING. Gates are expected to grow in one direction but permanent gaps are anticipated. If both LOW and HIGH are missing the COVERAGE list is rebuilt starting with the system-wide permanent_gaps list. (not yet implemented)

ACTIONTASK
: Name of Task that will cause the gate associated product to be updated to at least part of the range wantlow to wanthigh. The task is initiated by receipt of a ticket with action 4 or 5. The task initiation is done by the task manager.

PROJECT
: Name of the overall project supported by the gate. Serves to identify which permanent gaplist to use for instance. E.g. “MDI”, “HMI”, “AIA”, or “NA”.

COVERAGE
: List of non-missing data intervals. This list is the actual list for this

product and can be built from scratch by the STATUSCOMMAND
A maintenance task can compare the COVERAGE list and the permanent gap list to discover intervals of missing data that might be obtained.

COVERAGE_ARGS
A string containing any arguments needed to manage the coverage arguments that are specific to this gate/product. The special string “NEVER” prevents the coverage from being updated or consulted for this gate.

GAPLIST
List of permanent data gaps for this gate/product. This is initialized from the instrument gaplist and can be added to by manual intervention.

NEW_TICKETS
List of pending tickets linked to this gate.

CURRENT_TICKET
sequence number of the most recent ticket assigned to this gate.
The sequence number becomes part of the ticket name.

ACTIVE_TICKETS
List of tickets presently being acted upon by some task.

There is one gate for each program:product pair.
There may be multiple gates for a particular dataseries only so long as these multiple gates each service disjoint subsets of the series as specified by special prime-keywords. The name of the gate should indicate the subset as well as the product.

STATUS just queries the state of the gate.
ACTION pushes the gate to extend the open range.

##########################################################################

A gate is implemented as a directory.
new_tickets and active_tickets are subdirectories which contain new and active tickets.
See the Implementation directory/file list notes for details.

**GATEKEEPER**

The gatekeeper or ticket processor is a program that checks tickets and takes action as requested.

The gatekeeper maintains a list of all gates and periodically checks them for new tickets.

The gatekeeper keeps a list of active tickets and checks their status on a regular basis so that it can return the ticket when appropriate. A task with a pending ticket is usually a program in a wait state. It may be managed by a task manager that stops and starts tasks in response to tickets being returned.

Gatekeeper Program does:

A. For each Gate:

0. If GATESTATUS is HOLD, skip the gate.

0.5 If STATUSBUSY is set, skip the gate for this iteration since the STATUSCOMMAND is running and the state of the gate is unknown.

1. Update gate status if nextupdate time has come.

2. For each new ticket:

a. Test for expired ticket, set expired status if time out.
b. if ticket can be answered immediately, actions 1 or 2 or 3 (if done), just do it.
c. if ticket asks for passive wait (3), leave in queue with wait state until done
d. if ticket is type 4 or 5, start Action task and link ticket into queue.
This is done by moving the ticket into the active ticket queue then calling the taskmanager with the taskname, gatename, and ticketid. The taskmanager to call is named in a task attribute.

3. For each active ticket:

Test return state for action tickets, return if task finished. This is test of STATUS of tickets in a task done or failed list.

Only the gatekeeper will modify tickets and only task manager will modify task tables.

B. Sleep for a while

C. go to A.

**TASK**

A Task is a descriptor of a processing task that will update a dataseries.

- Tasks increase the open range of gates thus a task connects programs to data products with gates.
- Tasks issue tickets to test for or arrange for input data therefore tasks contain information needed to generate tickets.
- Tasks are represented by directories that contain sub-directories and
- files with keywords.

Task is (program -> data product -> gate ) actuator. I.e. specifies the rules to advance the gate that describes the state of a product made by a command.

A task has a fixed part which describes the task and information about how to compute the product. It also has a sub-descriptor for each instance of the task that is actuated.

All tickets originate in a task instance and are processed by a task, either the initiating task or another task.

Task overall descriptor is:

TASKNAME
: Name of task to put into Gate Action entry, should be useful for display. aka task_name. The TASKNAME my not contain a dash, “-“.

CURRENT_TASK
: Contains the TASKID if the most recently started instance of this task. TASKID is <taskname>-<date>-<sequence>

PRECONDITIONS
: List of ticket gates that must be satisfied for this instance of the task to be started. Implemented as list of gates with optional script called ‘prepare_ticket’ to be sourced.

ACTIVE
: List of active instances (./active/)

DONE
: List of completed task instances. (./done/) STATE tell success or failure.

RETAIN_TIME
: Number of days to retain DONE instances in the ARCHIVE-OK lists. Default is 1 day.

ARCHIVE-OK

ARCHIVE-FAILED
: Lists of completed instances after the gatekeeper has extracted the STATE information and returned the generating ticket.

STATE
: Number of active instances.

TARGET
: Gate that this task will update

COMMAND
: Command to make product from low to high

PARALLELOK
: Number to indicate parallel processing of sub ranges is OK. 0 means do each MAXRANGE chunk in serially, n means do in parallel but allowing at most n instances running at once.

MAXRANGE

Max range to compute in each instance of the task.

MANAGER
task manager that knows how to run this task.

Task instance descriptor:

TASKID
Formed from taskname, date, and sequence number for that date.

TICKET
ticket that caused the instance to be started.

STATE
0 = idle task is not active
1 = waiting task is waiting for tickets to be returned
2 = running task is active
3 = waiting for precondition tickets to be returned
>5 = error in task, some subtask had error

WANTLOW
WANTHIGH
Range requested for this task to satisfy, copied from the originating ticket.

PARENT
TASKID of parent task tells where the ticket should be returned to.

PENDING
List of tickets issued by this instance.

RETURN
List of tickets returned to this instance.

A task is initiated by an action ticket which specifies a gate range.
When a gate gets an action ticket the gatekeeper calls the task's manager to start an instance of the task. When the task instance is done, the task manager sets the task instance STATE 0 or 5 depending upon success and moves the task instance to the DONE or FAILED list.

The task instance is not done until all related processing is complete.

The task manger uses the task descriptor to:

1. submit precondition tickets
2. wait for all precondition tickets
3. run a command to make a product. If the coverage range of the ticket is larger than the max allowed, the task manager breaks the job into multiple segments, which are themselves tickets. Then it waits for those tickets to come back before returning the prime ticket to the calling task.

The system is initiated by issuing a getrange ticket for each gate.
Processing is initiated by a user initiating tickets for the desired end products.

NOTE: the logic is now:

if (wantrange > maxrange)
issue tickets to same gate for chunks of max size maxrange.

if action=4, which means only compute parts of product not already available, use the status command with range args to find the missing data chunks in the current range. For each chunk, issue a ticket to the gate with action==5, i.e. force the computation.

Finally, action==5 handling and computing:

```
    if any preconditions, issue ticket for each precondition and wait
    for completion.
    Then issue command for the current range.
```
Note that taskmanager calls itself to deal with range too large, small chunks of data to compute, and preconditions (different gate though). This is done by issuing tickets to its own gate or the precondition gates.

##########################################################################

Implementation:

Each task has a directory containing the static parts of the descriptor and a sub-directory for each active instance of the task. Each instance contains several directories. When a task COMMAND program is run, the working directory is the instance TASKID directory which is allowed to contain log files and temp files for the instance. The directory structure of tasks is described several pages below.

The preconditions handling needs some extra notes. The precondition list is implemented as a directory which itself contains a directory for each precondition gate. The names of these directories are the gate names of the prerequisites. These prerequisite gate directories themselves may contain a script called “prepare_ticket”. This prepare_ticket script will be “source”ed by the taskmanager just prior to calling maketicket.csh to issue a ticket to the precondition gate.
The current directory will be the precondition directory where the prepare_ticket file is when this happens so that other files in that directory may be used by the prepare_ticket sourced script. That ticket call will be to maketicket.csh and will have the form:

```
  maketicket.csh gate=<precondition gate> \
                 taskid=$taskid \
                 wantlow=$USELOW \
                 wanthigh=$USEHIGH \
                 action=$ACTION \
                 special=$SPECIAL
```

The user need not and should not change any variables used by the taskmanager except USELOW, USEHIGH, ACTION, and SPECIAL. The prepare_ticket script may do what ever is necessary to fix-up the default values prior to having the ticket issued. The defaults are set in the taskmanager just prior to calling the prepare_ticket script and are:

```
                set USELOW = $WANTLOW
                set USEHIGH = $WANTHIGH
                set ACTION = 4
                set SPECIAL = NONE
```

Note that WANTLOW and WANTHIGH are taken from the task that is being run to service the parent ticket. They are in the units and type of the task which has preconditions and may not be suitable for the precondition gate itself. for example a request for a synoptic chart for a given Carrington rotation, CARROT needs input data specified by date. So the default USELOW and USEHIGH will be set to some rotation number. The prepare_ticket script should convert these to dates suitable for the prerequisite gate. The variable TYPE contains the type of the WANTLOW and WANTHIGH values. The variable GATEDIR contains the path to the precondition gate directory which contains information about the target gate, such as the file $GATEDIR/type which contains the type needed for USELOW and USEHIGH.

Note that this is also the place to pass on other parameters via the SPECIAL argument. It is convenient to configure the value for SPECIAL as a set of param=value pairs with blanks between so that the precondition task can extract needed information. Obviously the prepare_ticket script and the target task must agree on the format.

The default ACTION will allow the use of existing data to fill the needed data range. ACTION=5 will force recomputation of the data for that window of the gate. In test gates/tasks ACTION=5 has been used to force pre-staging of DSDS data known to be available but perhaps offline.

The prepare_ticket script may need helper variables. To minimize the chance for harming the taskmanager which is also a csh script in present form these prepare_ticket scripts should use only all-caps symbols and avoid altering:
  TYPE, WANTLOW, WANTHIGH, QUITTING, STATUS, FAILURE, EXITOK

Some variables are preset and may be used but must not be changed. These include:
  WORKFLOW_ROOT, WFDIR, NOT_SPECIFIED, TASKID, TYPE, GATEDIR

The ARCHIVE-DONE and ARCHIVE-FAILED lists should be purged periodically to remove old records. These are the task directories archive/ok and archive/failed.

**TASKMANAGER**

The task manager runs a task.

The task manager runs a command on behalf of the gatekeeper when
it gets an action ticket. It is not a daemon. It is an instance
of a standard program that runs in the task instance directory.
The task manager program is specified in the ACTIONTASK task associated with a
gate.
It will be one of perhaps a few programs that service different
types of gates.
A task manager instance is started by the gatekeeper.
It is initiated with arguments containing: gate=<gate> task=<task>
ticket=<ticketid>
Taskmanager must set the ticket status=5 upon error.
Taskmanager is called with "&" so it does not block the gatekeeper.

If the task manager instance needs to wait for one or more tickets
it will wait in a sleep/test loop.

A task descriptor specifies task max ranges to execute at one time.
The task manager can generate multiple instances of the task program
if the requested range exceeds the max range.

There may be a few standard task managers.

Manager does:

0. Create a new taskid
   taskid is <taskname>_<day>_<number>
   where day is yyyymmdd
   make with a call to GetNextID which will takes sequence
   number file as single arg, and will use 4 digit number.
   Thus, in creating task, make file containing fixed part, i.e.
   simply taskname. Name it "taskid"
1. enter task into running task queue
   this is done by creating and populating a directory in
   the task's <active> directory.
1.5 Link the originating ticket into the taskid directory.
2. if action==4 create action=5 tickets for gaps in data in wantrange
   then exit Uses same logic as step 3 below.
3. if wantrange> maxrange recursively call same task with tickets to gate
   a. create action ticket for subranges to satisfy range.
   b. submit action tickets to gate.
   c. wait for all tickets.
   d. done
   NOTE: if in-order processing is required must do subranges
   in order, else can do in parallel.
4. if wantrange less or equal to max range just do the job now
   a. foreach ticket in precondition list prepare a wait
      until ready ticket for this chunk.
   b. wait for all need tickets
   c. run command (via script that calls qsub) – finally do this task
      For now, assume that the command run here does all the work of
      setting up the drms_run and qsub scripts and waiting for them to finish.
   d. check exit status of qsub script
   e. update task instance state as done

```
    f.  done


A ticket is returned by moving it to the tasks ticket_return dir.
The gatekeeper is the only process allowed to change tickets once they are
created.  The taskmanager is the only process allowed to change task instance
STATE values.

###############################################################################

Implementation

Top level directory structure:

bad_gates/
  <gate_name>/          # place to put gates that cause errors in gatekeeper

gates/
  <gate_name>/
    gate_name           # file, one line, one word, <gate_name>
    product             # file, one line, one word, drms seriesname
    gatestatus          # file, one line, one word, ACTIVE or HOLD
    statusbusy          # file, no contents, exists when statustask is active
    sequence_number     # file containing single line with CURRENT_TICKET
    low                 # file, one line, one word, LOW value
    high                # file, one line, one word, HIGH value
    type                # file, one line, one word, "time", "sn", ...
    key                 # file, one line, one word, principal prime-key name
    lastupdate          # file, one line, one word, time
    nextupdate          # file, one line, one word, time
    updatedelta         # file, one line, one word, seconds
    statustask          # executable command to update low,high
    actiontask          # file, one line, one word, name of update task
    gaplist             # file, one line per data gap, pair of times per line
    coverage            # file, one line per data segment, pair of
                        # times per line.
    coverage_args       # additional arguments that may be needed for updating
                        # the coverage information.
    new_tickets/        # directory containing un-examined tickets
      ticket files...
    active_tickets/     # directory containing tickets currently being processed
      ticket files...   # by a statustask or action task.
tasks/
  <task_name>.../
    task                # file, one word, task name
    parallelOK          # file, one word, 0 or 1
    maxrange            # file, one word, value of max range in units of
                          LOW, HIGH, use seconds tor TYPE==time.
    target              # file, one word, gate that this task updates.
    note                # file, descriptive text for the task
    state               # file, one word, state of task indicating active or
                          idle.  Contains the number of active instances.
    taskid              # file, one word, taskname as seed, will contain
                          most recent taskid
    retain              # file one word, number of days to keep DONE instances.
    preconditions/
```

```
  <gate>...          # directories containing ticket information
    prepare_ticket   # optional file, script, optional script that can be
                       'sourced' to make the shell variables USELOW and
                       USEHIGH for a ticket to be made for the named gate.
command*             # executable command to process data from LOW to HIGH
                     # Uses wantlow and wanthigh and returns status=0 for OK
                     # change dir to the tasks/active/<taskid> directory
                     # before calling the program.  That dir contains needed
                     # info. 'command' is executable and is usually a link.
manager*             # taskmanager program to execute this task, usually
                       a link
active/
  <taskid>.../       # taskid is formed as <task_name>.<task_number>
    ticket           # file, link to initiating ticket, updated when
                       task done
    state            # file, one line, status of subtask
    subtasks/
      <taskid>...    # empty files named same as child tasks. May be absent
                       if no subtasks.
    pending_tickets/
      <ticketid>...  # empty files named same as tickets sent to gates
    ticket_return/
      <ticketid>...  # returned ticket files.
    parent           # contains "taskid" of parent task
    wantlow          # file, one word, target value, copied from <ticket>
    wanthigh         # file, one word, target value, copied from <ticket>
done/
  <taskid>.../       # completed tasks, waiting for gatekeeper inspection.
archive/
  ok/
     <taskid>.../    # completed tasks that have been examined by
                     # gatekeeper.  must be cleaned regularly.
archive/
  failed/
     <taskid>.../    # completed tasks that have been examined by
                     # gatekeeper.  must be cleaned regularly.
```

## Implementation programs/scripts

Several support programs have been made to implement the workflow capability. These are the basic active components:

### gatekeeper.csh

Gatekeeper is started with no arguments. It performs the gatekeeper functions by watching over the gates directories and moving tickets from queue to queue as needed based on values stored in the ticket files. When it is in the active part of its loop it creates (touches) an empty file named GATEKEEPERBUSY. At the end of each pass through all the gates it removes this flag and sleeps for 10 seconds. It also sets and watches for the file named “keep_running”. Removing “keep_running” allows a clean stop of the gatekeeper. gatekeeper is usually running. It is the primary ‘daemon’ in the workflow system. It is usually run as:

```
gatekeeper.csh >& log &
tail –f log
```

There are several flag filenames that are accessed to turn on debug babble to stdout.

### taskmanager.csh

Taskmanager is called with three arguments as:

```
taskmanager.csh task=<taskname> gate=<gatename> ticket=<ticketid>
```

The task is the ‘action-task’ that is invoked to process data in response to a ticket sent to a gate. Taskmanager checks the gate specifications to see if any preconditions, prerequisites, exist for products described by the gate and if so, the taskmanager generates tickets to make sure that those input products are available or that the conditions are met for the gate to be “advanced”. Once prerequisites are met the taskmanager executes the command that actually does the work. The action command is run with the task instance directory as its current directory and no command-line arguments. The action command is usually a script that can examine its detailed instructions contained in files task instance directory. The taskmanager will split a large computation into multiple sub-tasks governed by the MAXRANGE attribute of the task descriptor. It will run these subtasks either serially or in up to ‘parallelOK’ parallel groups. The taskmanager should check for action=4 and action=5. If action=4 then do the show_coverage that used to be done in the gatekeeper and issue action=5 tickets to itself do compute any missing subranges. The gate variable coverage_args contains a string (default “none”) that is appended to the show_coverage command line. The special string “NEVER” essentially converts any action==4 into action=5. This is needed if the product series is not slotted.

### cleanup.csh

Cleanup is a simple script that examines each task for DONE instances greater than that task’s RETAIN_TIME number of days. The script may be run as a cron job or as a gate/task pair itself. The old task instances are removed from the archive/ok directories.

### watchgates.csh
### checkgates.csh
### watchtasks.csh
### checktasks.csh

These simple scripts show the user that status of selected the gates and tasks. It is a simple text display that is updated during the sleep cycles of the gatekeeper. It will not start when GATEKEEPERBUSY exists They will terminate when the KEEP_$USER_watching_<gates|tasks> file is removed.
A much better watcher function should be built. If a gate status is HOLD, skip both the gate and the associated task printing.

### maketicket.csh

Maketicket is a script that takes the attributes of a ticket from its command line and generates a ticket file that is properly formatted. The ticket is placed into the specified gate. It is called with its parent taskid and target gate. All tickets are "created" by a task instance and delivered to a target gate. Maketicket is called with taskid, gate, wantlow, wanthigh, action, and special as <name>=<value> arguments
in any order. If 'taskid' is not present it is assumed that the ticket is generated by the virtual task instance called "<task>-root" which will be the repository for completed task instances initiated by the user or sometimes program submitted tickets without a source taskid.

### wait_ticket.csh

Wait_ticket takes a single argument of a ticketid and waits until processing on the ticket has ended. If the ticket was for action=4 or 5 then the ticket is printed along with the path to the task instance where the processing was done. An example use is:

```
wait_ticket.csh = `maketicket.csh gate=suphil.vwVtest \
      wantlow=1996.05.17_09_TAI wanthigh=1996.05.17_09:04_TAI`
```

### cancel_ticket.csh

Cancel_ticket tries to find and remove a ticket and its progeny recursively. It looks in several places and kills any manager processes called with the ticket specified.

### GetNextID.c

Presently in ~phil/jsoc/proj/myproj/apps. GetNextID takes a filename of the current ID file, and an optional "fixedpart" string which contains the preface of a new ID value. the ID value is <fixed_part>-<date>-<number> where the fixed_part can be a string like <taskname> and <date> will be the date wehre the next ID is assigned, and the <number> will be a unique sequence number within that date. This is very similar to the VSO cart ID and the JSOC RequestID format. The binary is in WORKFLOW_ROOT/bin.

The structure building tools:

### makegate.csh

Makegate creates a gate directory structure with details as specified on the command line. The full list of expected parameters is: gate_name, gatestatus, product, low, high, type, key, lastupdate, nextupdate, statustask, actiontask. The default updatedelta is one day, the default low and high are NaN, The default nextupdate is the current time so that the gate is immediately in need of update. The default statustask is scripts/status-general.csh witch is usually OK. gate_name, product, type, key, and actiontask must all be

specified.  Actiontask is the name of a task as made by maketask.csh while statustask is the full path to an executable script or program.  Gatestatus contains the word HOLD or ACTIVE and is created in HOLD state.  To allow the gatekeeper to begin using the gate and associated task, run the “enablegate.csh” script.

## maketask.csh

Maketask creates a task directory structure and fills in specifiec from information given on its command line.  It is called with param=value pairs where the params expected are:  task, parallelOK, maxrange, target, note, state, command, and manager.
ParallelOK defaults to 0, state to 0, manager to taskmanager.csh.  At least task, target, command, and maxrange must be given.  command and manager must be full paths to the script/program that computes the data product and the taskmanager to use respectively.  A symbolic link is made to the given names and that link is used when the taskmanager and command are invoked.

A test set of gates and tasks which can be used as an example:

## init_pipeline.csh – should not be run in a working workflow system

Init_pipeline deletes and existing gates and task directories.  To build and start the system from scratch one can go to $WORKFLOW_ROOT, remove tasks and gates directories, then run:

```
  init_pipeline.csh
  csh add_XXX # run some gate/task creation scripts.
  gatekeeper.csh &  # in a dedicated window or capture its output.
  watchover.csh &  # in a dedicated window
  enablegate.csh gatea gateb …  # enable all gates needed
```

Once the status is stable, the user can drop in new tickets with calls to maketicket.csh and the ticket will be processed to make the desired product.

### enablegate.csh

Enablegate simply sets that gatestatus flag to ACTIVE allowing the gatekeeper to begin processing tickets sent to it.  Call enablegate.csh with a list of gates to enable.

### pausegate.csh

Pausegate simply sets the gatestatus flag to HOLD causing the gatekeeper to ignore the gate and not examine new tickets or pending tickets.  Gates are created in HOLD state and must be enabled to start things going.

**Steps to add a processing step to the workflow.**

In order to add a new product/gate a gate must be made and an action-task must be made. These can be made with calls to e.g. makegate.csh and maketask.csh. The notes below describe the information and scripts or programs that should be made or identified to be able to call the above scripts. In general for each data product to be maintained there must be one gate, one task, one status script and one update script. The (more recent) “add_XXX” scripts creates all of these with a single csh script. The status scripts usually are one or two calls of show_info while the update script calls a computational DRMS module.

0. Examine one of the test scripts, e.g.: $WORKFLOW_ROOT/add_suphil.tests.csh

1. To make the gate, identify the output product. Need:
   product=seriesname
   type=time or type=sn, type of prime key, ‘time’ or ‘sn’
   key=<primekeyname>, name of prime key that describes the product.
   actiontask=<name of task to be made in step 7 below>
   statustask=<name of STATUSCOMMAND in step 2 below>

2. Identify or create STATUSCOMMAND that does at least
   find first and last 'time'. A general purpose STATUSCOMMAND is at
   scripts/status-general.csh. STATUSCOMMAND must “rm statusbusy” when it is complete. The status script must be executable.

3. Identify or create the update COMMAND (in Task) that will
   update the product. This command will be run in
   the <taskid> directory with wantlow and wanthigh targets that it must read and the ticket file from which it can get other parameters.
   The “COMMAND” should not return until the computation is complete or has stopped with an error. It may place log files
   in the current directory so long as their names do not collide with workflow reserved names in the
   $WORKFLOW_ROOT/tasks/<task_name>/active/<taskid>
   directory described above. This script will usually call qsub and may often call drms_run to accomplish the processing. It is responsible for waiting until all qsub initiated processing is complete before returning. It should set an exit status to 0 if all is OK and non-zero in case of fatal errors.

4. Identify the TaskManager program to use for the above task
   program. Use the standard one, $WORKFLOW_ROOT/taskmanager.csh until we find cases that it does not work for.

5. Make the gate.
   Need:
   Name of gate should indicate the product, no dashes.
   UPDATEDELTA - maximum time before checking product status
   Info in step 1 above.
   Make gate using makegate.csh.

6. Precondition list.
   Need list of gates/products that are input to the update COMMAND.
   Need gate names and any SPECIAL info to be passed to the update script.

7. Make the "Action task" that will be used by the gatekeeper to lookup the taskmanager appropriate for this product. The action_task should be made by a call to the maketask.csh script.

Need:
Name the "Action task", should indicate the gate being supported.
Do not use dashes in the name.
Precondition ticket list, you will make a directory for each precondition with the directory name the same as the precondition gate name. If there are any changes to the default precondition ticket, ACTION=4 and no SPECIAL arguments needed for the update COMMAND you will need to make a file named “prepare_ticket” in the precondition gate directory. This file will be “sourced” by the taskmanager after ticket defaults have been set but before the maketicket call. Variables you can change are ACTION, USELOW, USEHIGH, and SPECIAL. The precondition ticket will be issued with: wantlow=$USELOW wanthigh=$USEHIGH action=$ACTION special=$SPECIAL.
SPECIAL = string to be read by COMMAND command when started

8. If the update COMMAND is a script the first lines need to do something like:

```
set TASKWD = $cwd # if you will change dirs in the script.
foreach ATTR (TYPE WANTLOW WANTHIGH)
  set ATTRTXT = `grep $ATTR ticket`
  set $ATTRTXT
end
set SPECIAL = `grep SPECIAL ticket`
foreach SPECIALWORD ($SPECIAL)
  set $SPECIALWORD
end
```

this will leave shell variables containing TYPE, WANTLOW, WANTHIGH and all SPECIAL key=value pairs needed for the command. (e.g. in the example code fragment below, if SPECIAL contains “alph=abcdefg” then $alph would contain abcdefg.)

The command should return a status=0 if the WANTLOW-WANTHIGH range was satisfied, else an error code.

If the script uses qsub to start the script it may do the following to manage the wait for completion. E.g.:

```
set HERE = $cwd
set CMD = $HERE/qsubcommand
echo “#! /bin/csh –f” >$CMD
echo “cd $HERE       >>$CMD
echo “command_to_run from=$WANTLOW to=$WANTHIGH alph=$alph” >>$CMD
echo “echo $? > $HERE/procstatus” >>$CMD
echo “rm –f $HERE/qsubID” >>$CMD
echo `qsub –e $HERE/qsublog –o $HERE/qsublog –q j.q $HERE/$CMD` > qsubID
while (-e $HERE/qsubID)
    sleep 10
end
set STAT = $HERE/procstatus
exit $STAT
```

with variations as needed.

**Notes on implementation of RepeatTask functionality**

By making two gate/task sets the repetitive product capability can be built without any new base functionality. for example if there are a set of products to be computed each 10 minutes, the following could be assembled:

1. Clock_gate/Clock_task – combination provides a virtual data product whose "high" value is the current time, or an offset time. Gate responds to tickets with action=3, which means wait until condition is met. The associated status task simply sets gate high to the current time. Then an action=3 ticket will simply wait until the ticket's wanthigh time has arrived.

2. Production gate/task – This pair gets a ticket sent to it for each interval of time to process some products. All the products that need to be updated at the cadence for this pair will have tickets issues in the update task. So the first thing is to issue the precondition ticket to the clock gate which will cause a wait until after the desired interval has passed. After the time has been reached, the action task is called. The action task can be a script that does two things. First it issues a "don't wait" ticket to its own gate for the subsequent interval. This sets up the next interval. It will not block processing so that if processing has stopped for a while, it can be caught up quickly. Second the action task creates and submits tickets to the desired product gates to actually compute the products for the current interval. A wait must be done for these products to complete before exiting the action task script. See the script "add_repeat_hmi_nrt" as an example of a script that sets up processing two products at a 10-minute cadence.

3. To initiate the processing, simply issue a ticket for the first interval desired to the repeat gate, "repeat_hmi_nrt" in the example.

4. But be careful that the maxrange on the repeat task must be huge so that if the repeat task is called first with a large interval, for instance as after a processing down time, that task is not split into multiple threads each of which will march on carrying more and more buckets of water … Let the processing task deal with the large range. Splitting non-repeat tasks into parts is harmless and in fact allows flow control.

**PROBLEMS, CONCERNS, etc.**

the pending_tickets directories should probably be common and in the task dir vs the active instance dir. Then the limit on parallel_ok would work for all tickets in that task instead of only in the local instance. Change needs some verification that no damage. For instance does the ticket that makes more tickets live in a pending_ticekts dir, probably, so they should not be counted. Perhaps need another layer, one of commands active. For now user be careful to not flood d02.

no, that does not work. for action=4 tickets pending is tickets back to the tasks gate which will then recursively call taskmanager. this case is where need to have good pending count.

But in action=5, where all tasks end up, the pending list is for preconditions which are not sent to this same task. (maybe should somewhere be a test for this infinite loop case…

18 june 2010

today all of the scripts using ‘date’ were modified to always work in UT instead of local time. There was a mix causing some of the chaos in multiple sets of tickets issues.

end of day, huge lags in DRMS/SUMS caused some repeated processing requests to issue overlapping tickets. Better management of gap filling is high priority.

19 June 2010

Perhaps tickets should be registered with the origination task and kept there with sufficient info to track down where they are.

Also, perhaps all action==5 tickets should contribute to a maintained coverage chart, showing blocks in process where there is already an action 5 ticket issued. Then at the end of that process, the coverage map gets updated. Of course this is just a show_coverage query…
But there needs to be some way, before issuing new action==5 or actually in the process of action==4 tickets, to know what is already being done.

## EXAMPLES

soon…