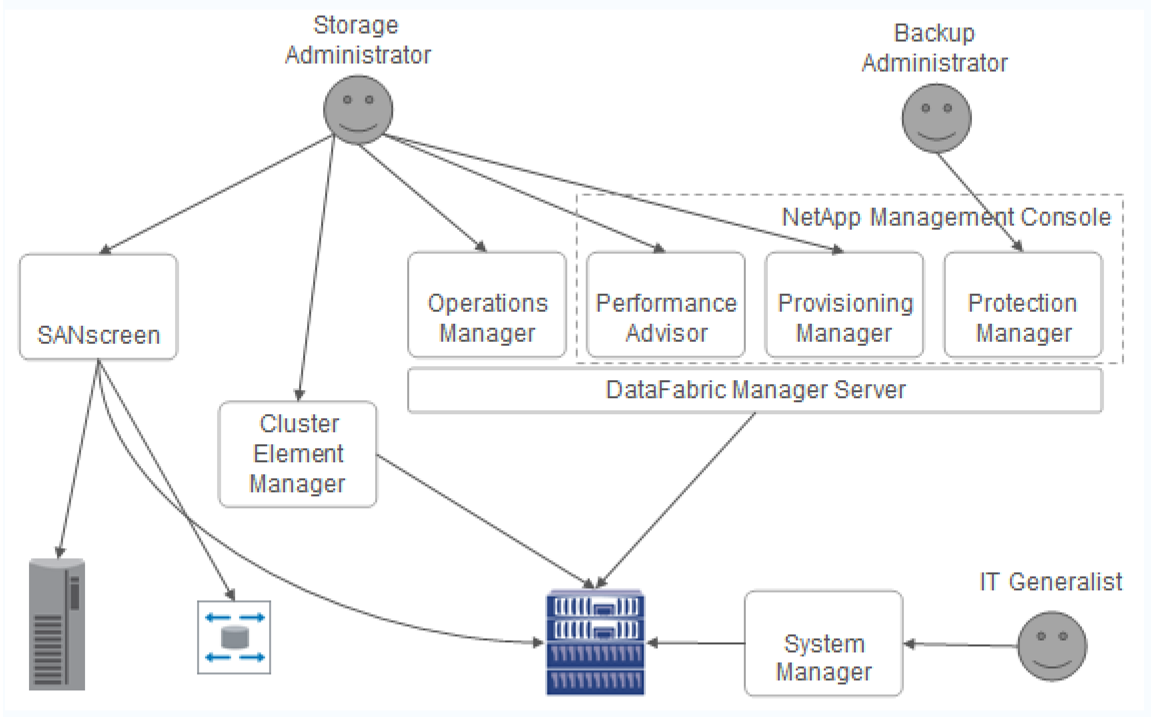
DFM architecture description

Prepared by Sahu & Pradeep

NetApp Management Apps

This diagram shows how DFM exists within the overall landscape of NetApp management applications:



I stole this diagram from these articles. The first one gives a great background on the management applications:

NetApp Storage and Data Management, Part 1

The second one describes future product directions:

NetApp Storage and Data Management, Part 2

DFM System Overview

The DataFabric Manager (DFM) system provides application interfaces and supporting functions to monitor and manage a network of NetApp appliances. Some characteristics of DFM:

- DFM is an optional component, not required to operate any NetApp appliance.
- DFM runs on a management station separate from the NetApp appliances, and does not require that addition of any software to the appliances being managed by DFM.
- DFM functions can be accessed by either Command Line Interface (CLI) or web-based GUI application.
- DFM integrates with FilerView.
- DFM supports multiple versions of Data ONTAP.

The primary goals of DFM are:

- Make system administrators more efficient when they use NetApp appliances.
- Relieve administrators of some mundane tasks, so that they can focus on higher level tasks.

In order to achieve these goals, it is important that DFM place a high priority on ease of installation/configuration/use.

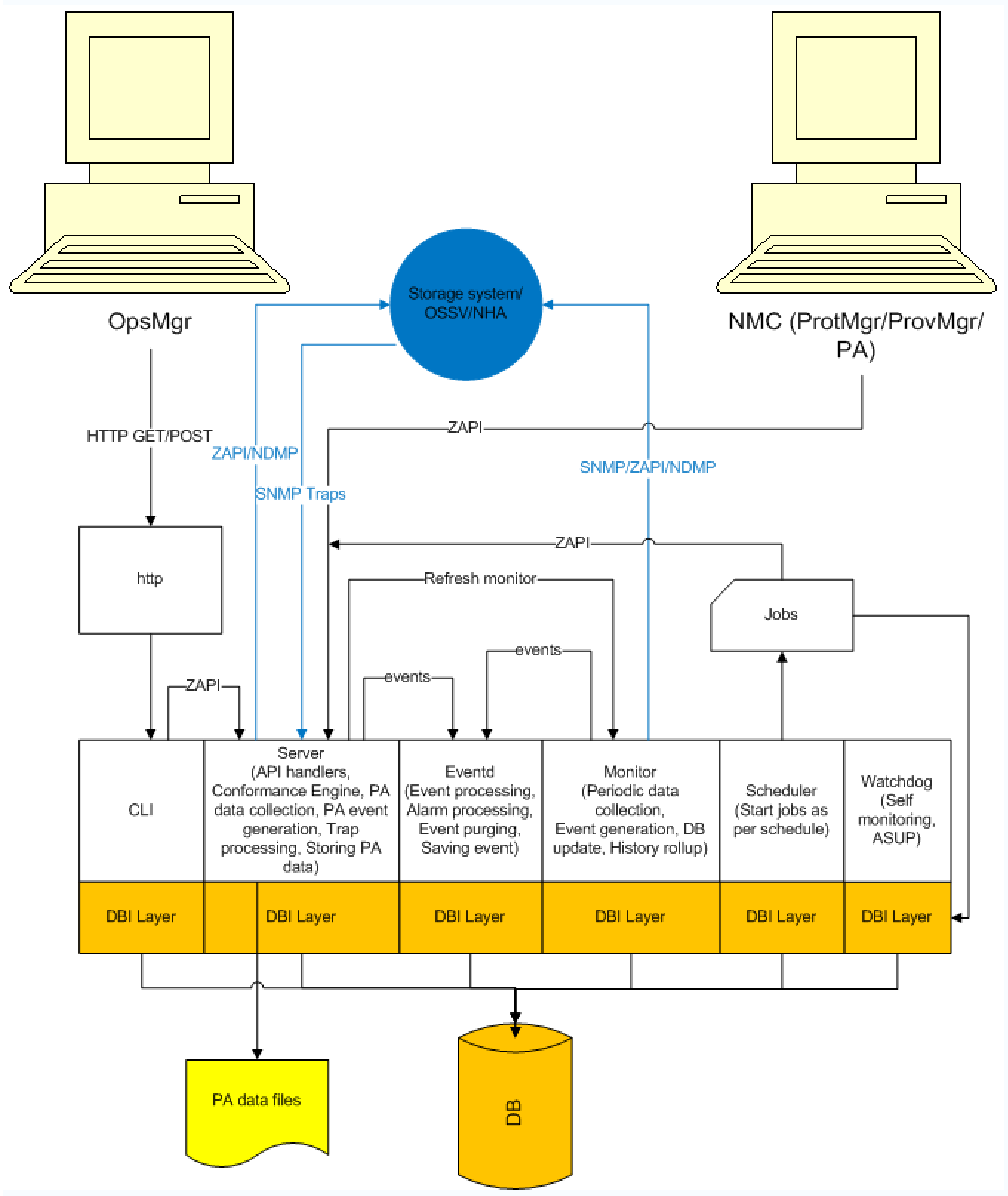
DFM performs automatic discovery via SNMP of filers, caches, networks, volumes, disks, etc. This configurable discovery process can be restricted to certain networks, decreased/increased in frequence, or even turned off once initial discovery is complete.

DFM periodically monitors the health/capacity/performance of filers, caches, volumes, etc. and saves the resulting data into a database. This data is made available for display/reports. Monitor data can be aggregated over time for graphing and planning purposes. The monitor process also generates user alarms based on configured threshold values.

DFM supports the assignment of volumes and hosts into customer-meaningful groups which can then be the subject of management functions that would normally have to be applied to each individual entity.

System Architecture

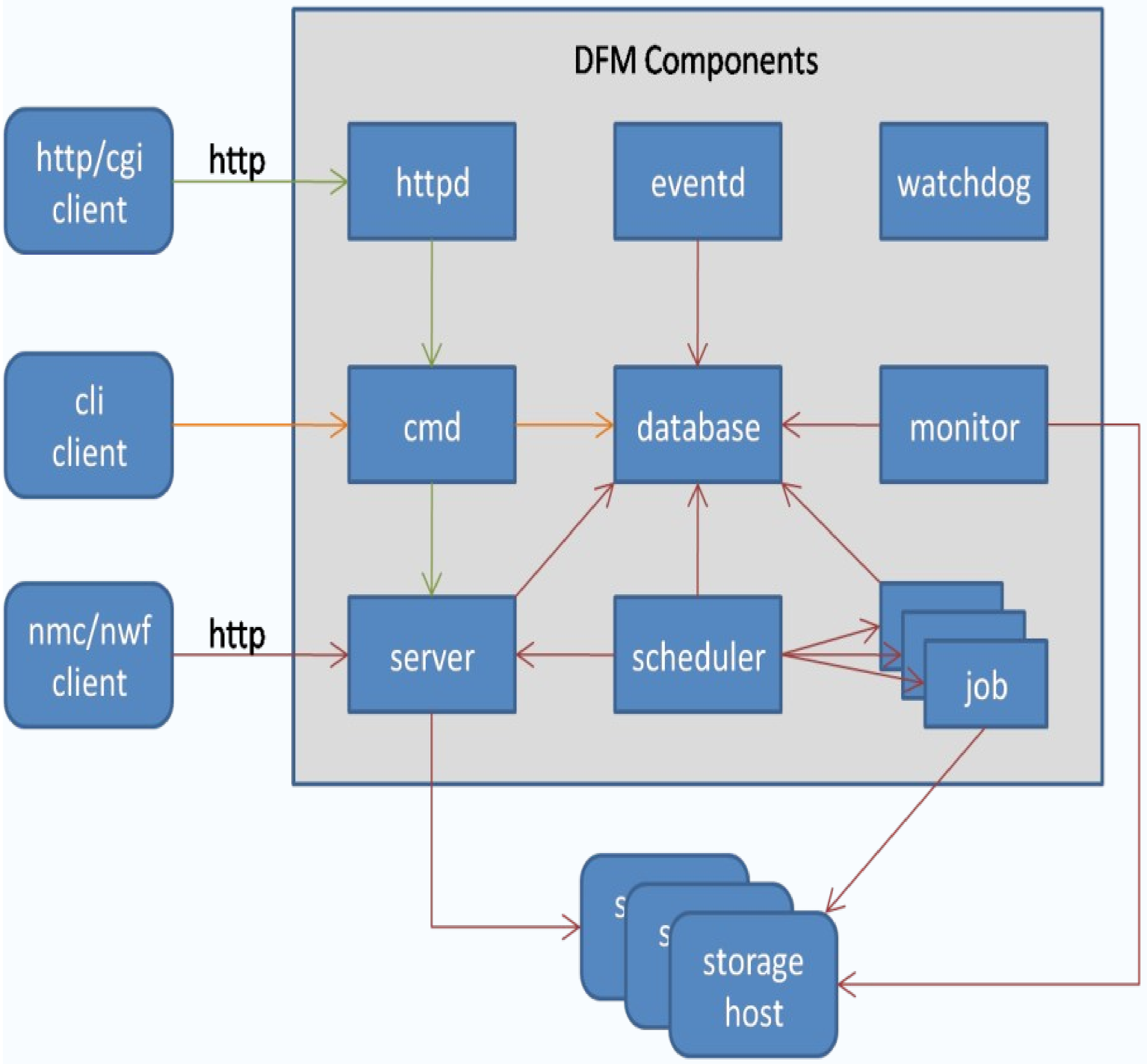
This diagram illustrates the overall architecture of DFM:



OpsMgr
NMC (ProtMgr/ProvMgr/PA)
Storage system/ OSSV/NHA
HTTP GET/POST
ZAPI/NDMP
SNMP Traps
ZAPI
SNMP/ZAPI/NDMP
ZAPI
http
Refresh monitor
Jobs
events
events
ZAPI
CLI
Server (API handlers, Conformance Engine, PA data collection, PA event generation, Trap processing, Storing PA data)
Eventd (Event processing, Alarm processing, Event purging, Saving event)
Monitor (Periodic data collection, Event generation, DB update, History rollup)
Scheduler (Start jobs as per schedule)
Watchdog (Self monitoring, ASUP)
DBI Layer
DBI Layer
DBI Layer
DBI Layer
DBI Layer
DBI Layer
PA data files
DB

Major System Components

This diagram illustrates the major components of DFM:



These DFM components can be organized into 3 categories:

- **Storage Hosts** - Storage hosts that are monitored/administered by DFM.
- **DFM Server** - Component of the server, most running in their own OS process.
- **DFM Clients** - User facing client interfaces to DFM services.

First, a description of the DFM server processes:

- **Database** - Relational DB, holds almost all data used by the DFM application system. Current database is embedded Sybase iAnywhere.
- **Monitor** - Polls storage systems to gather data and update the database. Generates events based on defined thresholds. Multi-threaded, with multiple monitors for different object types.
- **Eventd** - Event daemon that receives events from the monitor. Notifies user of events via configurable alarms.
- **Scheduler and jobs** - Performs scheduled background tasks such as: SnapVault updates, DB backups, report generation, config file updates, Snapshot jobs, etc.
- **Server** - Called the DFM Server - confusing because the whole system of server components is also called DFM Server. Primary purpose is to handle ZAPI requests. Also serves as a central controller for other subcomponents. Many threads run under this process.
- **Cmd** - CLI command processing. Older command implementations access the database directly, while newer implementations make calls to the DFM server process.
- **Httpd** - Embedded Apache server. Serves static HTML requests and dispatches CGI requests.
- **Watchdog** - Reports statistics and status of DFM services.
- **Core Services** - Not visible in the diagram, but there is a large set of core services that are utilized by the other server components.

Now, the DFM client types:

- **NMC/NWF** - The older NetApp Management Console or newer NetApp Web Framework. These clients send XML-formatted ZAPI requests via HTTP to a well-known socket on the server.
- **CLI** - Command Line Interface. These clients call one of several "dfm" executables and pass command line arguments to define the request.
- **HTTP/CGI** - These clients submit CGI requests to the HTTP server. CGI scripts convert the request to CLI invocations.

DFM implements the functionality of these products:

- **Operations Manager** - Access to monitor-collected data and admin of groups/policies for managed objects. Is this correct?
- **Protection Manager** - Functions related to data backup/recovery, SnapMirror, SnapVault.
- **Provisioning Manager** - Functions related to storage provisioning
- **Performance Advisor** - Collects performance statistics and what else?

Runtime Model

These are the DFM direct entry point executables. Except where indicated these run as services in their own process:

- httpd (http server - apache)
  - Embedded apache web server with HTTP and CGI support
- dbserver (db server - sybase)
  - Embedded Sybase iAnywhere relational database server
- dfmserver (src\server\main.c)
  - ZephyrAPI(ZAPI) interface to DFM functions
- dfmcmd (src\cmd\main.c)
  - Command Line Interface(CLI) to DFM functions
  - Does not run in its own process, is invoked by other processes
- dfmmonitor (src\monitor\monitor.c)
  - Monitor
- dfmeventd (src\eventd\eventd.c)
  - Event daemon
- dfmscheduler (src\scheduler\scheduler.c)
  - Starts scheduled jobs, each forked in its own process
- dfbm (src\backup\main.c)
  - Performs scheduled jobs related to data backup
- dfpm (src\datamgt\Datamgt.cpp)
  - Performs schedued jobs related to data protection
- dfmwatchdog (src\monmon\monmon.c)
  - Watchdog

DFM Server

Startup

The DFM Server is started as a service/daemon. When started, here are some significant things that happen:

- Run the executable server\main.c\main()
  - Call server\main.c\main_initialize()
    - Call server\server.cpp\server_startup()
      - **Call Performance Advisor initialization - libperf\perf.cpp\perf_init()**
        - Create/start one main control thread
          - Runs libperf\perf.cpp\perf_main_thread()
          - At intervals, checks performance data directory to make sure there is available space
        - Create/start/detach one host poller control thread
          - Runs libperf\perf.cpp\perf_host_poller_thread()
          - At intervals, determines which hosts need to be polled and dispatches each host to a thread from a zworkq
          - The poller zworkq initially contains 10 threads
            - Each thread runs libperf\perf.cpp\perf_poller_func()
            - Poller work thread collects the info and returns to the single poller control thread
            - Single poller control thread performs all the data updates when the worker threads have all completed
      - **Call ZAPI Server initialization - libserver\Server.cpp\Server::init()**
        - Call libserver\Server.cpp\Server::zapi_server_init()
          - Call libnetapp\na_zapi.c\zapi_server_new()
            - Create new ZAPI server context - libnetapp\na.h\zapi_server_t struct
              - Define the URL and port for receiving ZAPI requests
              - Define the list of supported ZAPI methods
              - Define libnetapp\na_zapi.c\na_zapi_handler() as the ZAPI handler function
          - Call libnetapp\na_zapi.c\zapi_add_handler_array()
            - Define the ZAPI method handler map in the zapi_server_t struct
            - All ZAPI methods map to function libzapid\zapid.c\zapid_method_handler()
        - Call libzapid\zapid.c\zapid_new()
          - Create new ZAPI server daemon context - libzapid\zapid.h\zapid_s struct
          - Set the previously created zapi_server_t as the server
          - Call libzapid\httpd.c\httpd_new()
            - Create new HTTP daemon context - libzapid\httpd.c\httpd_s struct
            - Create/start one HTTP listener thread
              - Runs libzapid\httpd.c\httpd_listener_func()
              - The listener does no real work, just blocks waiting for a new connection and then dispatches each new connection to a thread in the connection handler zworkq
            - Create a handler zworkq initially containing 2 threads
              - Each thread runs libzapid\httpd.c\httpd_connection_func()
              - Handler thread stays alive handling requests until the client closes the connection or we detect an unrecoverable error
              - For each request, search map for the defined handler function, then call the handler function

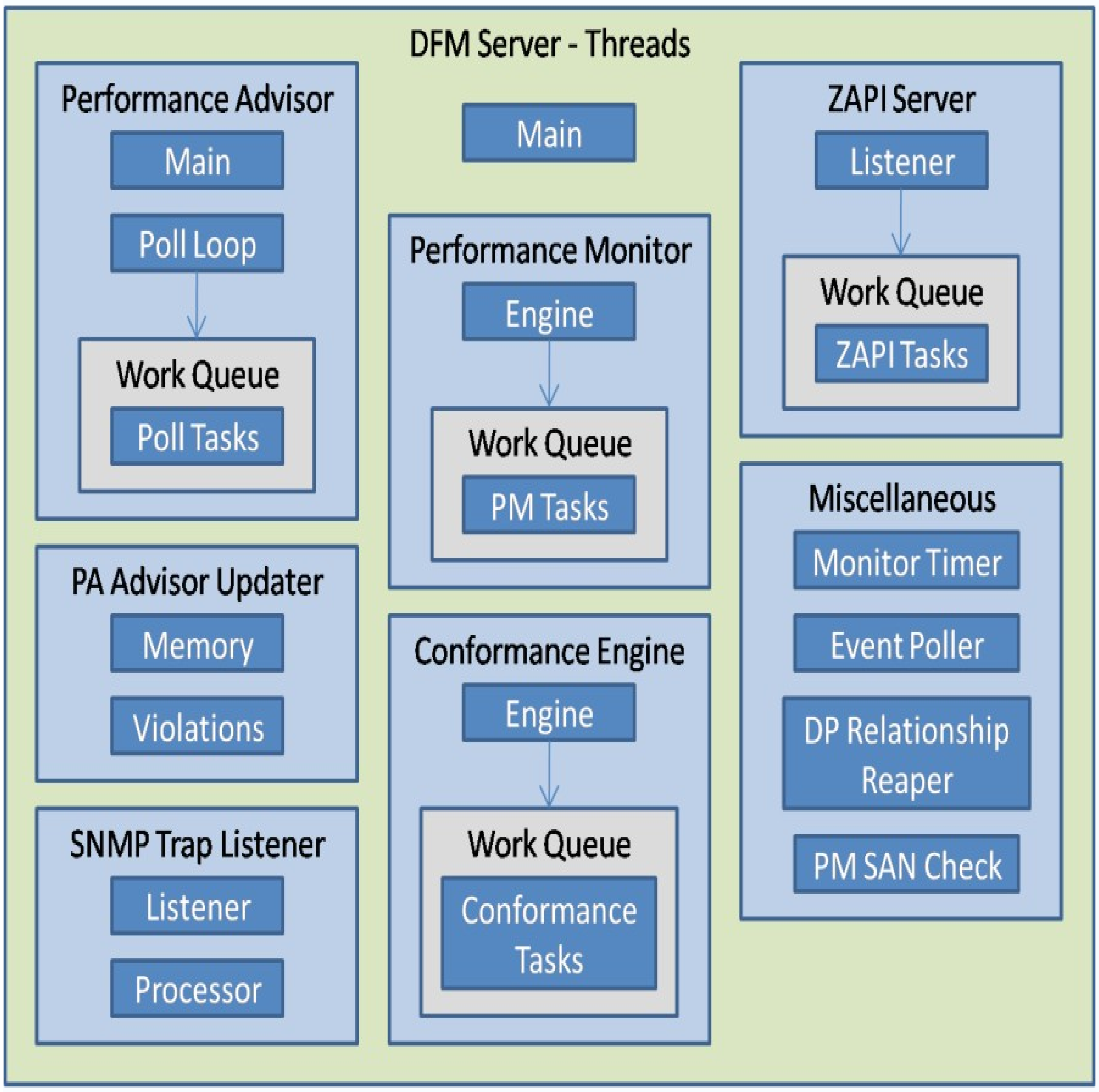
    - Call libzapid\httpd.c\httpd_register_handler to set the handler function for all requests
      - Handler function is libzapid\zapid.c\zapid_method_handler()
  - Call libzapid\zapid.c\zapid_set_dispatch_func()
    - Set libserver\Server.cpp\Server::zapi_dispatcher() as the dispatcher function in the zapid_s struct
  - Call libzapid\httpd.c\httpd_set_parallelism() with max threads count
    - Increases zworkq for HTTP handler threads to 32
  - Call libserver\Server.cpp\Server::listeners_init()
    - Call libserver\Server.cpp\Server::add_listeners() for HTTP
      - Call libzapid\httpd.c\httpd_add_listener()
        - Bind an HTTP listener to the configured port
    - Call libserver\Server.cpp\Server::add_listeners() for HTTPS
      - Call libzapid\httpd.c\httpd_add_listener()
        - Bind an HTTPS listener to the configured port
- **Call Performance Monitor initialization - libmonitor\pmjob.c\pmjob_force_init()**
  - Create/start/detach one monitor control thread
    - Runs libmonitor\pmjob.c\pmjob_todo_engine()
  - Create a monitor job zworkq initially containing configured number of threads
    - Each thread runs libmonitor\pmjob.c\pmjob_run_engine()
- **Call Trap Listener initialization - libserver\traplistener.cpp\trap_start_trap_listener_thread()**
  - Create/start Trap Listener thread
    - Runs libserver\traplistener.cpp\trap_listen_for_traps()
      - Bind to port udp:162 to listen for SNMP traps
      - Register libserver\traplistener.cpp\trap_snmp_input() as the callback function
        - Adds trap to a queue for the Trap processor
      - Call libserver\traplistener.cpp\launch_trap_processor()
        - Create/start Trap Processor thread
          - Runs libserver\trapprocessor.cpp\trap_processor_thread()

            - At intervals, checks trap queue and processes any traps

- **Call Event Poller initialization**
  - Create the singleton instance of libevt\EventPoller.cpp\EventPoller
    - Create an instance of libevt\EventPoller.cpp\EventPollerThread
  - Start the EventPollerThread
    - Run libevt\EventPoller.cpp\EventPoller::poll()
      - At intervals, checks the database for new events and notifies EventSubscribers
- **Call Monitor Proxy initialization - libdfmcore\Monitor.cpp\monitor_proxy_init()**
  - Create a number of libdfmcore\Monitor.hpp\MonitorProxy objects with callback functions
  - Create the Monitor Timer, instance of class libadtpp\Timer.hpp\Timer
    - Wakes at defined interval and runs the registered TimerCallback objects
  - Add libadtpp\Timer.hpp\TimerCallback objects to the Monitor Timer
  - Start/detach the Monitor Timer thread
- **Call Conformance Engine initialization - libdatamgt\ConformanceEngine.cpp\ConformanceEngine::init()**
  - Create a conformance task zworkq initially containing 24 threads
    - Each thread runs libdatamgt\ConformanceEngine.cpp\conf_engine_process_task()
  - Create/start a single libdatamgt\ConformanceEngine.cpp\ConformanceEngineThread object
    - Calls libdatamgt\ConformanceEngine.cpp\ConformanceEngine::dispatcher()
      - Wakes whenever a change is made to the conformance task queue
        - Task queue contains subclasses of libdatamgt\DPTask.hpp\DPTask
      - Dispatches conformance tasks to the task zworkq
- **Call DP Relationship Reaper initialization - libdfmcore\DpRelationship.cpp\DpRelationshipReaperThread::reap_relationships()**
  - Create/start instance of libdfmcore\DpRelationship.hpp\DpRelationshipReaperThread
    - At intervals, reaps stray libdfmcore\DpRelationship.hpp\DpRelationship objects
- **Call SAN Check Thread initialization - libdfmcore\PMSANChecks.cpp\SANCheckThread::run()**
  - At intervals, checks Protection Manager write guarantees
- **Perform Performance Advisor Updater initialization**
  - Create/start an instance of libperf\PerfThreshMemUpdater.cpp\PerfThreshMemUpdater
    - Runs libperf\PerfThreshMemUpdater.cpp\PerfThreshMemUpdater::run()
      - At intervals, check defined memory thresholds
  - Create/start an instance of libperf\PerfThreshViolationUpdater.cpp\PerfThreshViolationUpdater
    - Runs libperf\PerfThreshViolationUpdater.cpp\PerfThreshViolationUpdater::run()
      - At intervals, consolidates memory violation history

Components

The DFM Server consists of a number of different components, all of which run as threads inside of the single DFM Server process. This diagram shows the components and threads. Each of the darker blue boxes represents a single thread or a pool of threads in a work queue.



ZAPI Request Processing

This is the usual path for processing of a ZAPI request by the DFM Server:

- Client builds HTTP/XML ZAPI request data
- Client submits request to defined DFM server HTTP listener socket
    - Request connection received by the HTTP listener thread in libzapid\httpd.c\httpd_listener_func()
    - Builds a libzapid\httpd.c\httpd_connection_s struct
    - Dispatches the connection to a thread in the connection handler zworkq passing the httpd_connection_s struct
        - The HTTP handler executes libzapid\httpd.c\httpd_connection_func()
        - Calls libzapid\httpd.c\httpd_handle_one_request() passing the libzapid\httpd.c\httpd_s struct
            - Builds the libzapid\httpd.h\httpd_request_response_t struct and populates the request data
            - Calls libzapid\zapid.c\zapid_method_handler() passing the HTTP request/response struct and the libzapid\zapid.c\zapid_s struct
                - Performs username/password authentication
                - Calls libnetapp\na_zapi.c\zapi_execute() passing the libnetapp\na.h\zapi_server_t struct and the HTTP request body, which contains the XML ZAPI request data
                    - Calls libnetapp\na_zapi.c\zapi_execute_so() passing the same arguments
                        - Builds a ZAPI api context libnetapp\na.h\zapi_t struct
                        - Calls libnetapp\na_zapi.c\na_zapi_handler() passing the zapi_t struct and the request XML data
                            - Parses the ZAPI XML request into a character buffer and also a chain of libnetapp\na.h\na_elem_t structs and packages both into a libnetapp\na_xml.h\na_xml_ctx_t struct
                            - Calls libnetapp\na_zapi.c\na_zapi_dispatch_and_log() passing the zapi_t struct and the na_elem_t struct chain representation of the ZAPI request
                                - Calls libserver\Server.cpp\Server::zapi_dispatcher() passing the zapi_t struct
                                    - Calls the actual ZAPI api handler function passing the zapi_t struct
                                - Calls libnetapp\na_zapi.c\na_zapi_log_result() passing the zapi_t struct
                                    - Log the ZAPI api results
                - Puts the zapi_execute() return data into the HTTP response struct
            - Send the HTTP response on the socket

        - Handle any more requests
    - End the connection

DFM Monitor

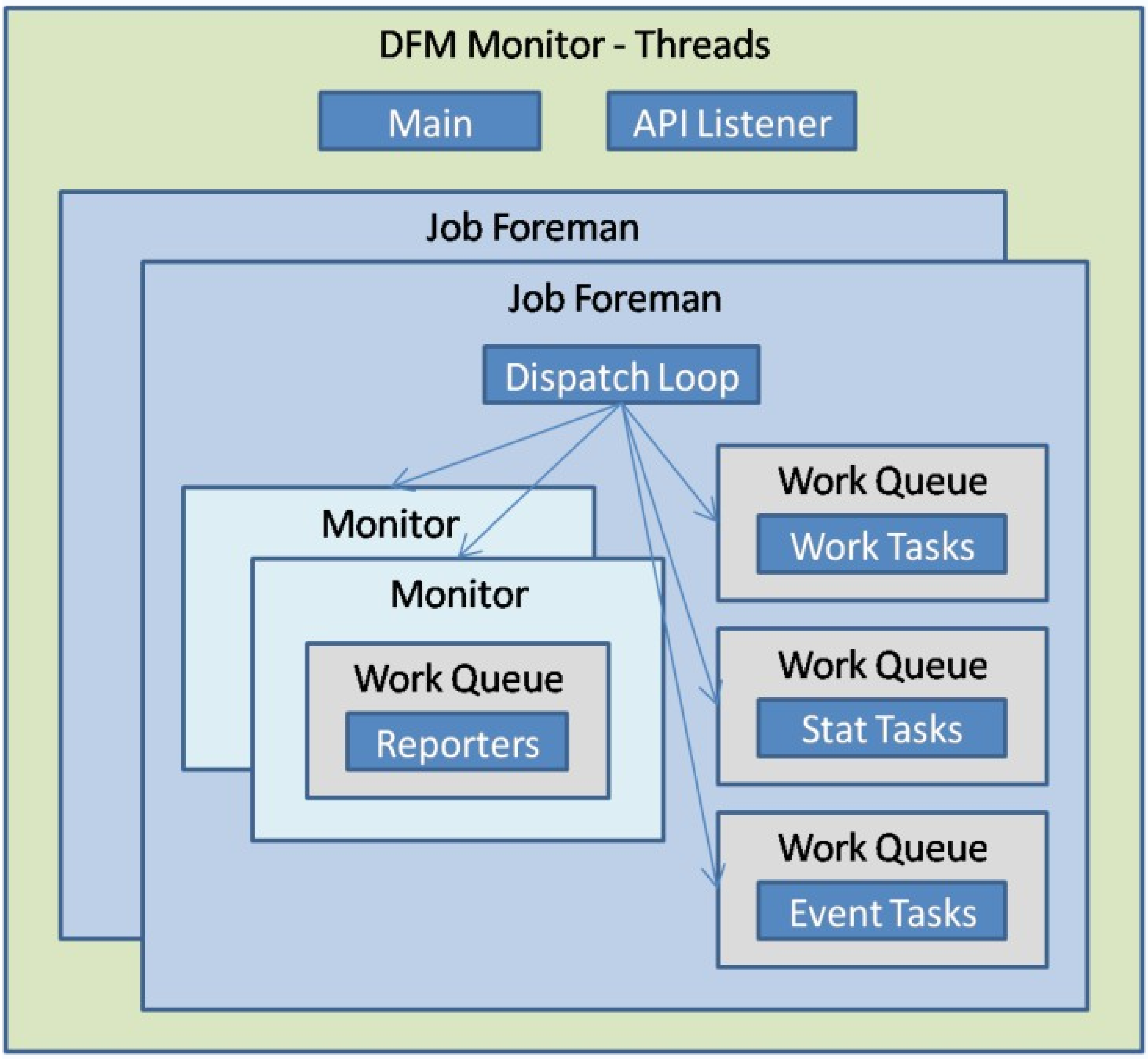
Startup

The DFM Monitor is started as a service/daemon. The DFM Monitor is responsible for collecting data from the storage hosts and writing this data to the database. This task is implemented via an abstract processing model that consists of Job Foremen and Monitors. When the DFM Monitor is started, here are some significant things that happen:

- Run the executable monitor\monitor.c\main() which calls real_main()
    - **Perform static initialization - monitor\monitor.c\mon_static_init()**
        - For each monitor type supported by DFM, designated by a libmon\mon.h\mon_job_funcs_t struct, defined in the hard-coded MON_MAP #define located in libmon\mon.h
            - Call the optional static init function at mon_job_funcs_t.f_static_init
    - **Capture live system data - monitor\monitor.c\mon_refresh_global_data()**
    - **Create the job foremen - monitor\monitor.c\mon_new_foremen()**
        - For each monitor type supported by DFM (mon_job_funcs_t)
            - Identify the foreman configured to handle this monitor type, by looking in the list of configured job foreman, designated by a monitor\monitor.c\mon_foreman_opts_t struct, defined in the hard-coded mon_foreman_opts array located in monitor\monitor.c
            - Call monitor\monitor.c\mon_add_monitor()
                - Build the monitor\monitor.c\mon_monitor_t struct
                - Create a workq containing a configurable number of reporter threads
                    - Each thread runs monitor\monitor.c\mon_reporter()
                - Add the new monitor struct to a temporary array that will later be attached to the identified foreman
        - **For each configured foreman (mon_foreman_opts_t) for which there are monitors**
            - Create a monitor\monitor.c\mon_foreman_t struct
            - Assign the foreman configuration options (mon_foreman_opts_t) to the foreman
            - Assign the array of monitors (mon_monitor_t) to the foreman
            - Create a workq containing a configurable number of worker threads
                - Each thread runs monitor\monitor.c\mon_worker()
            - Create a workq containing one bean counter thread
                - Each thread runs monitor\monitor.c\mon_bean_counter()
            - Create a workq containing one event sender thread
                - Each thread runs monitor\monitor.c\mon_event_sender()
            - Create/start the single job dispatcher thread for this foreman
                - Runs monitor\monitor.c\mon_foreman_func()
    - **Register the single "monitor/running" API via libdfm\api.c\api_register()**
        - This API call resolves to run monitor\monitor.c\mon_is_monitor_running()
    - **Create/start the single Monitor API listener thread via libadt\zthread.c\zthread_create()**
        - Runs monitor\monitor.c\mon_listener_thread()
    - **Enter infinite loop waiting for exit termination**
        - At intervals of 60 seconds, call monitor\monitor.c\mon_refresh_global_data()

Components

This diagram illustrates the components present in the DFM Monitor at runtime. The darker blue boxes are the running threads:



Foreman Dispatcher Loop

Each foreman has a job dispatcher thread that runs monitor\monitor.c\mon_foreman_func(). This function loops forever, looking for jobs and then dispatching the found jobs to worker threads. Each foreman owns a number of monitors. Each subclass of monitor has unique functions to find work, perform jobs, etc. The functions for each monitor type are defined in the libmon\mon.h\mon_job_funcs_t struct.

Here is the typical processing sequence for one cycle through the mon_foreman_func() loop:

- **Call monitor\monitor.c\mon_foreman_findwork()**
    - For each monitor(mon_job_funcs_t) owned by this foreman:
        - Call monitor\monitor.c\mon_monitor_findwork()
            - **Call <mon_job_funcs_t>.f_findwork()**
                - **Query the database to determine jobs that need to be performed**
                - For each required job:
                    - Call monitor\monitor.c\mon_new_job()
                        - Call libmon\mon.c\<mon_job_creator>()
                            - Create a job libmon\mon.h\mon_job_t struct
                - Return an array of mon_job_t
            - Add jobs (mon_job_t) to the monitor
- **Call monitor\monitor.c\mon_foreman_dispatch_jobs()**
    - Process all jobs for all monitors
    - Loop through all monitors, handling only one job from each
    - Repeat loop through monitors until no jobs remaining.
    - For each monitor(mon_job_funcs_t) owned by this foreman:
        - Get the first job ready to be run in this monitor
        - **Dispatch to one of the foreman worker threads - monitor\monitor.c\mon_worker()**
            - **Call <mon_job_funcs_t>.f_start>()**
            - **Assign the job to one of the monitor reporter threads - monitor\monitor.c\mon_reporter()**
                - **Call <mon_job_funcs_t>.f_report()**
                - If any events are pending for the job:
                    - **Assign the job to one of the foreman event sender threads - monitor\monitor.c\mon_event_sender()**
                        - Send each of the events to the Event Daemon
            - Close this job - monitor\monitor.c\mon_job_free()
                - **Assign the job to one of the foreman stat counter threads - monitor\monitor.c\mon_bean_counter()**
                    - Log job statistics
                    - Destroy the job

        - Take an optional sleep delay - monitor\monitor.c\mon_foreman_sleep()
- Take an optional sleep delay - monitor\monitor.c\mon_sleep_if_necessary()

Scheduler

Startup

The Scheduler is started as a service/daemon. The Scheduler is responsible for running jobs that need to occur on some timed schedule. Each job is invoked as a CLI command running in its own process.

When the Scheduler is started, here are some significant things that happen:

- Run the executable scheduler\scheduler.c\main() which calls real_main()
    - For each scheduler type supported by DFM, designated by a libsched\dfmsched.h\sched_job_funcs_t struct, defined in the hard-coded SCHED_MAP #define located in libsched\dfmsched.h
        - **Call the optional initialization function at sched_job_funcs_t.sched_start**
        - Call the mandatory find work function at sched_job_funcs_t.sched_work
    - **Enter infinite loop waiting for exit termination**
        - For each scheduler type supported by DFM
            - **Call the mandatory find work function at sched_job_funcs_t.sched_work**
        - Call scheduler\scheduler.c\sched_sleep_until()
    - For each scheduler type supported by DFM
        - **Call the optional cleanup function at sched_job_funcs_t.sched_finish**

Find Work Example

Here is the typical processing sequence for the find work function of a specific scheduler type, using libsched\backup.c as an example:

- Start with a call to libsched\backup.c\backup_schedwork()
    - **Run any jobs already scheduled in the database - libsched\backup.c\backup_run_scheduled_jobs()**
        - Select jobs from the database - libdfm\dbi.c\dbi_selectall_stab()
        - For each job to be started:
            - Start the job - libjob\job.c\job_start()
                - Determine the appropriate CLI program name based on the job type - libdfm\dfmutil.c\dfmutil_get_program_from_name()
                - Build argument list to pass to the CLI program:
                    - "job", from #define CMD_JOB in libdfm\names.h
                    - "run", from #define CMD_RUN in libdfm\names.h
                    - <job_id>
                - **Invoke the CLI command in a new process - libadt\proc.c\proc_spawn2()**
                    - CLI processing eventually calls <job_funcs_t>.j_handler, which is:
                        - libjob\backup.c\bjob_handler(), defined in static struct backup_job in the same file
    - **Create/run any secondary volumes that require backup now**
        - Call libsched\backup.c\backup_start_one() for each one
            - Calls libjob\job.c\job_submit()
                - Write new job to the database - libjob\job.c\job_store()
                - Start the job - libjob\job.c\job_start()
                    - Same sequence as above
    - **Reschedule to wake in one hour - libsched\backup.c\backup_sched_next_hour()**

Processing

The Scheduler runs only a single main thread. This thread awakens at intervals or can be signalled to awaken. Once awakened, each of the defined job scheduler types is allowed to search for jobs that need to be run and then start those jobs. Each job runs in a separate process. Within each dedicated job process, type-specific job handling is invoked via a CLI command.

Event Daemon

Startup

The Event Daemon is started as a service/daemon. The Event Daemon is responsible for handling events generated by other DFM components. Each event is written to the database and, if the event triggers an alarm threshold, then notifications are dispatched as configured for the event/alarm type.

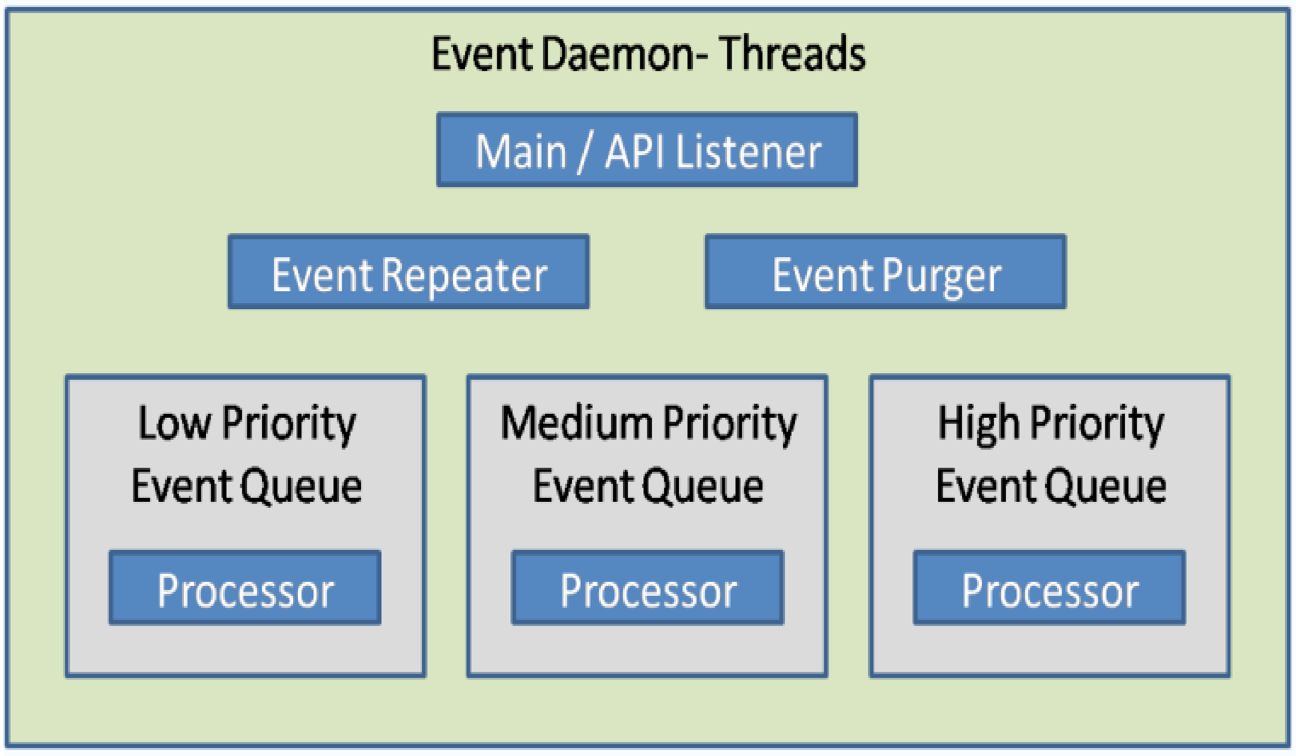
When the Event Daemon is started, here are some significant things that happen:

- Run the executable eventd\eventd.c\main()
    - **Event Daemon initialization - eventd\eventd.c\eventd_initialize()**
        - **Initialize the event queues - eventd\eventd.c\eventd_init_eventq()**
            - Call libdfm\eventq.c\eventq_new() to create 3 separate event queues for low/med/high priority
        - **Spawn event handling threads - eventd\eventd.c\eventd_spawn_threads()**
            - Call libadt\zthread.c\zthread_create() to create 3 event processor threads for low/med/high queues
                - Each thread runs eventd\eventd.c\eventd_processor_thread()

            - The single thread handles all events for the queue
        - Call libadt\zthread.c\zthread_create() to create a single event repeater thread
            - Thread runs eventd\eventd.c\eventd_repeater_thread()
            - Wakes at intervals to send any repeat event notifications that are required
    - **Register Event Daemon supported APIs - libdfm\api.c\api_register()**
        - event/post
        - alarm/modified
        - event/update_app_status
- **Start the API listener - eventd\eventd.c\eventd_listener_thread()**
    - This method runs on a thread in windows, but runs inline on other platforms
    - When the API listener runs as a thread, the main thread does nothing other than waiting for the listener thread to complete
    - Initialize the API listener sockets - eventd\eventd.c\eventd_init_serv_sock()
    - **LOOP FOREVER**
        - **Wait for and handle a single API request - eventd\eventd.c\eventd_process_select_fd()**
        - Start the event purger thread if needed - eventd\eventd.c\eventd_purger_thread_required()
            - Event purger thread runs eventd\eventd.c\eventd_purger_thread()
    - Clean up

Components

This diagram illustrates the components present in the Event Daemon at runtime. The darker blue boxes are the running threads:



Watchdog

The Watchdog is a very lightweight process that runs a single thread. At a configured time interval, which defaults to 5 seconds, the Watchdog wakes and performs the following steps for each process that is configured to be watched:

- Test that the process is running
- Collect processing stats
- Check for and report spikes in processing stats, based on configured thresholds
- Report all stats at configured time interval
- Autorestart failed process if not running

Curently the following processes are watched:

- Watchdog
- Monitor
- DFM Server
- Scheduler
- Event Daemon
- Database Server

CLI Processing

There are 4 distinct DFM CLI executable modules, each implementing a subset of the supported DFM CLI commands. These are the modules and their main() function locations:

- dfdrm - disaster\main.c\main() - disaster recovery functions
- dfbm - backup\main.c\main() - backup functions
- dfpm - datamgt\Datamgt.cpp\main() - Protection Manager functions
- dfm - cmd\main.c\main() - remaining functions

The commands implemented within each CLI module are organized into a hierarchy. The module name is considered to be the single abstract root command, with many primary commands descended from the root and some number of secondary commands descended from each primary command. In general, the code supports an open-ended command hierarchy with many levels. I'm not sure that we currently have any commands below the secondary level.

Each command, regardless of its position within the command hierarchy, defines a number of properties, arguments, and options. The typical command invocation syntax looks like this:

<module> <primary> <secondary> [args/opts]

example: dfbm job run 1234

Data Structures

An array of libcmd\opt.h\opt_options_t structs is used to define a set of command options.

The libcmd\opt.h\opt_command_t struct is used to define the overall properties of a single command. Each opt_command_t struct includes, among other things, these significant fields:

- oc_id - Unique numeric identifier of the command
- oc_name - Command name used in the invocation string
- oc_options - Array of opt_options_t structs to define options for this command
- oc_subcommands - Array of opt_command_t structs to define hierarchical children of this command
- oc_handler - Handler function for this command
- oc_parent - Hierarchical parent of this command

For all CLI modules, the file that contains the main() function also contains:

- A single anonymous enum of supported command/subcommand combinations
- A single opt_command_t struct, named cmd_toplevel, for the abstract root command
- Multiple opt_options_t struct arrays, where each array may be shared by a number of related commands
- Multiple opt_command_t structs for the module's primary commands, collected into a single array with varying names per module, attached to the root module command via the oc_subcommands field
- Multiple opt_command_t struct arrays, one per primary command, where each struct in the array defines properties of a single secondary command, attached to the primary command via the oc_subcommands field

Command Processing

In all 4 of the CLI modules, the main() function is a simple shell that calls an entry point within libcmd. The dfm, dfbm, and dfdrm modules call libcmd\cmd.c\cmd_main(), while the dfpm module calls libcmd\cmd.c\cmd_main_zapi(). Both of these methods simply call the underlying function libcmd\cmd.c\cmd_main_internal() with different values for a boolean argument which indicates whether to pass the DB connection into the command handler function.

The cmd_main() function is for legacy CLI implementations, so it allows the DB connection to be passed to the command handler and then the handler is free to perform direct DB queries. The cmd_main_zapi() is for newer CLI implementations and does not pass the DB connection to the command handler. This is to ensure that the handler implementation is based on ZAPI calls, without making any direct DB queries.

Arguments passed from the module's main() down into cmd_main_internal() include:

- Command line arguments
- Module name
- Module root opt_command_t struct

Given those initial arguments, here is the processing sequence of cmd_main_internal():

- Parse command and associated args/options - libcmd\opt.c\opt_parse()
    - Also identifies the associated opt_command_t struct for the specific command
- Obtain a DB connection for the command handler - libcmd\opt.c\opt_db_connect()
- Obtain a ZAPI server connection for the command handler - libcmd\user.c\user_get_serverapi()
- **Call the appropriate handler function from opt_command_t.oc_handler**
- Record an audit log of this command - libcmd\cmd.c\cmd_audit_log_cli_action()
- Produce command output - libcmd\cmd.c\cmd_print_results_and_exit()