Distributed Crawler Application

General architecture specification, phase A

The DC application is multithreaded server class software. Inter process interactions uses ZMQ TCP sockets connections server and client type. In process interactions uses ZMQ inproc sockets connections servers and clients. Asynchronous inproc resources access as well as requests processing provided by MOM-based transport provided by ZMQ sockets technology. No system or POSIX access controlling objects like mutexes or semaphores used. The only one kind of processing item is message. The several messages containers types like queues, dictionaries, lists and so on can be used to accumulate and organize processing sequences. The key-value DB engine is SQLite.

There are two listened by DTM application TCP ports:

1. Administration.
2. Client.

Administration – handles requests from tool client applications. Requests of that type fetches statistical information from all threading objects as well as can be used to tune some configurations settings of DTM application directly at runtime period.

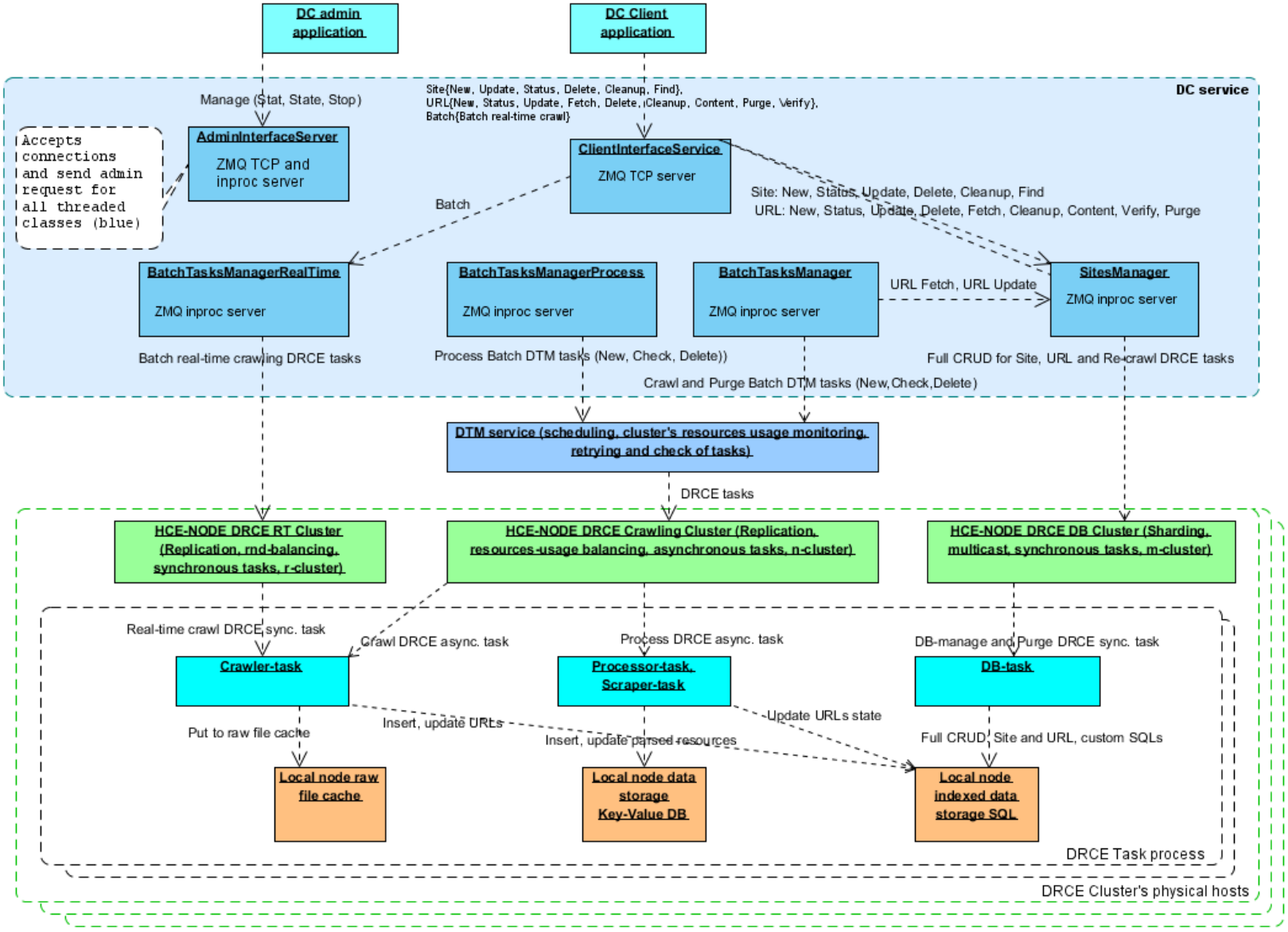
Client – handles requests from client side API or tool application. Requests of that type are main functional queries for tasks and related with HCE DRCE interaction protocol.

All requests are represented by json format of message. The structure of requests and main tasks logic principles are separated from concrete execution environment. This gives potential possibility to use some several execution environments engines, but not only HCE DRCE cluster.

Object model

All active networking usage objects (blue primitives on architecture UML diagram) are based on threaded class object. Also, all threaded objects uses list of transport level items of two types: client and server. Both are based on ZMQ sockets, server type can be TCP or inproc. TCP used for external connections, inproc – for all internal inter-threading events interactions

Application architecture model

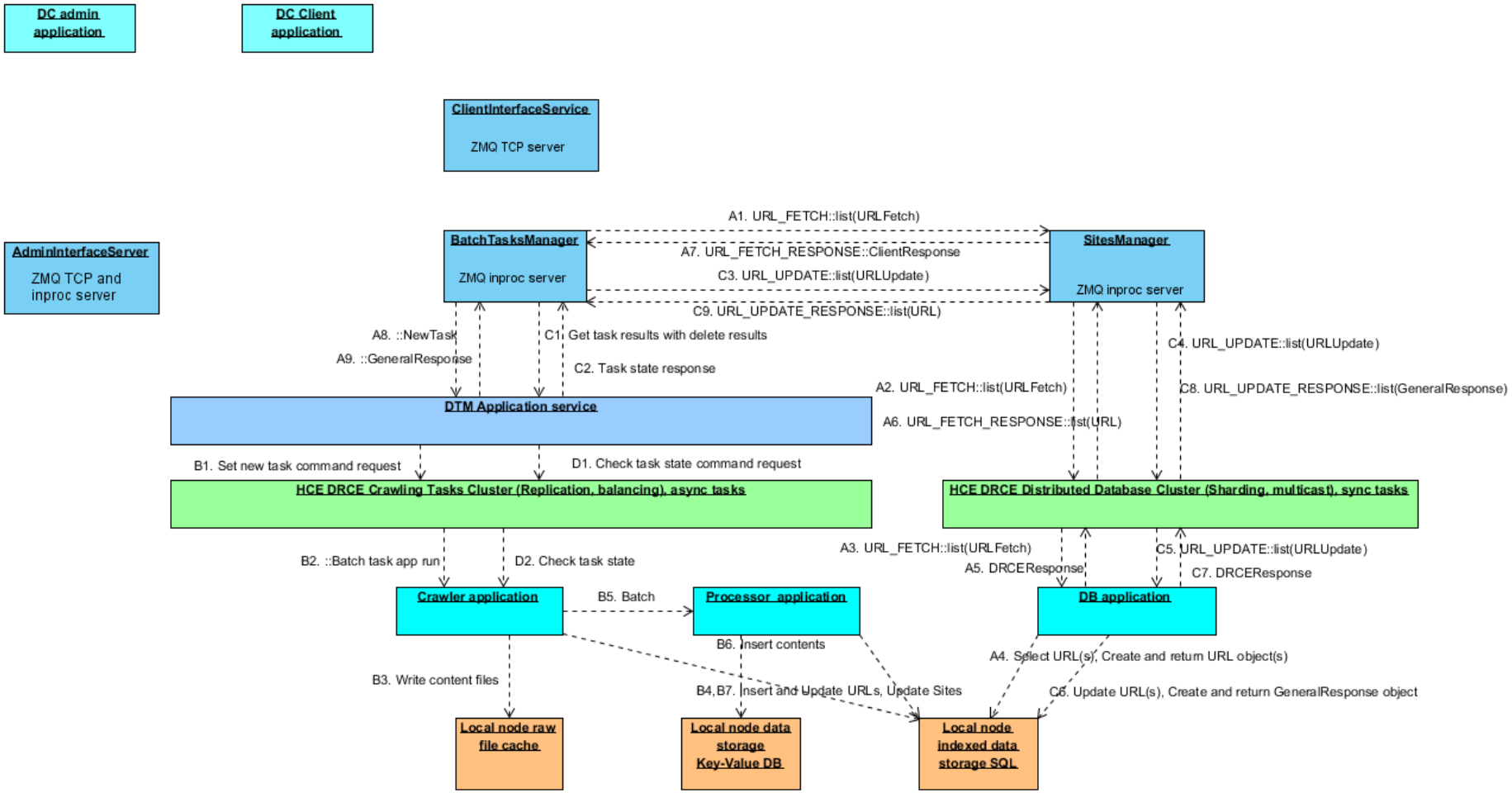


Operations

Operations are supported inside the service as isolated actions and available via client interface. There are different objects like Site, URL and Batch can be involved in to the algorithm of operation execution as well as it can to have more deep and long results than immediately returned in response.

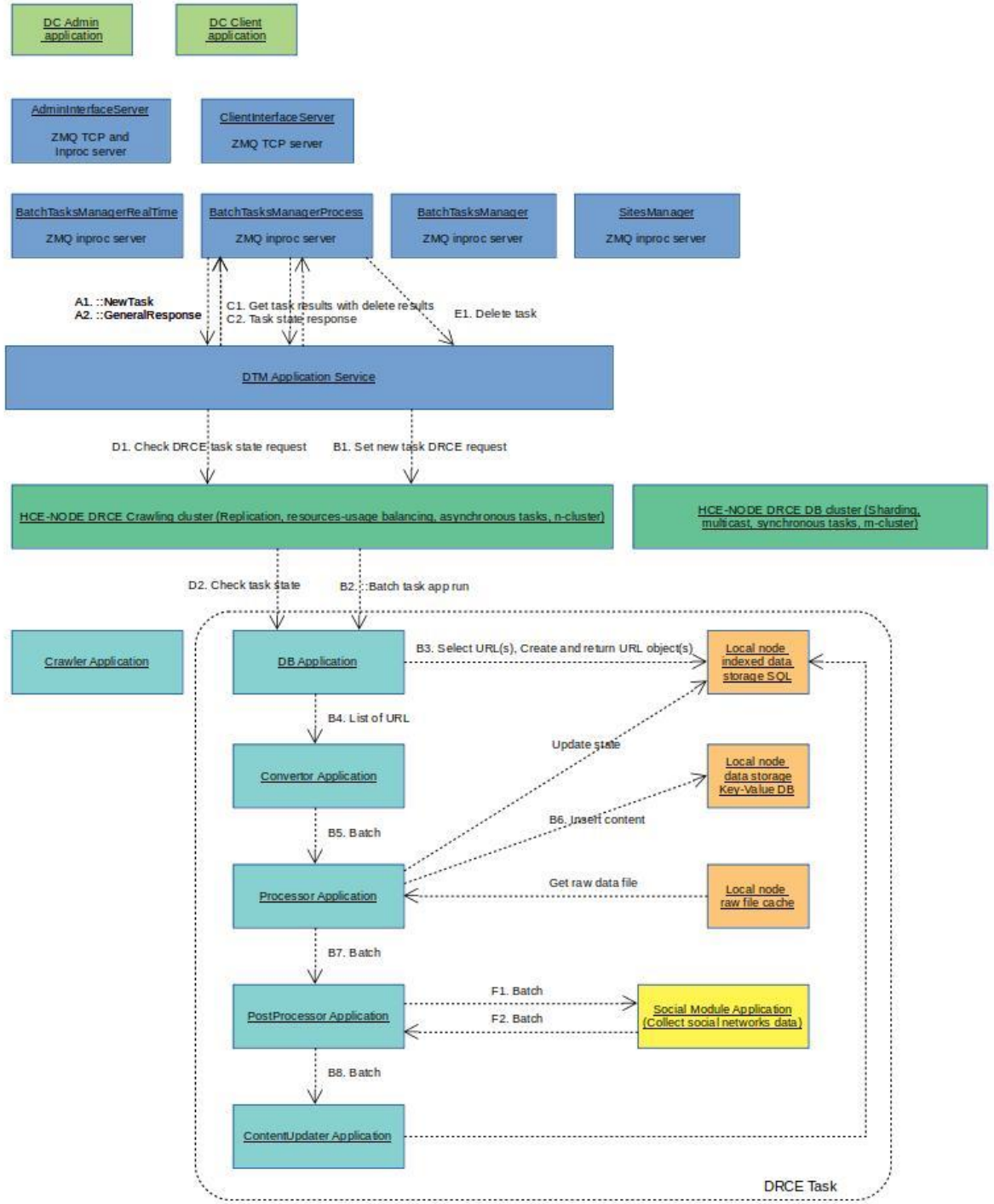
The regular crawling

The regular crawling is periodic process that creates the Batch object by requesting the lists of URLs object from the SiteManager. The list of URLs is unique and only not present in another active batch tasks are accepted. The Batch objects sending in to the DTM service as tasks and managed by the tasks queue. After finish of task execution URLs from the Batch are updated in state CRAWLED(4) at all data hosts. Before task execution all URLs from the Batch are send to insert as new with state SELECTED_TO_CRAWL(2) to all data hosts – the URLs propagation method. The DTM service set the Batch task on n-type cluster and its router uses resource usage balancing mode to choose proper data host to execute. This way URLs crawled only on one data host from several, but collected on all with state CRAWLED(4) to prevent farther redundant crawling and processing.



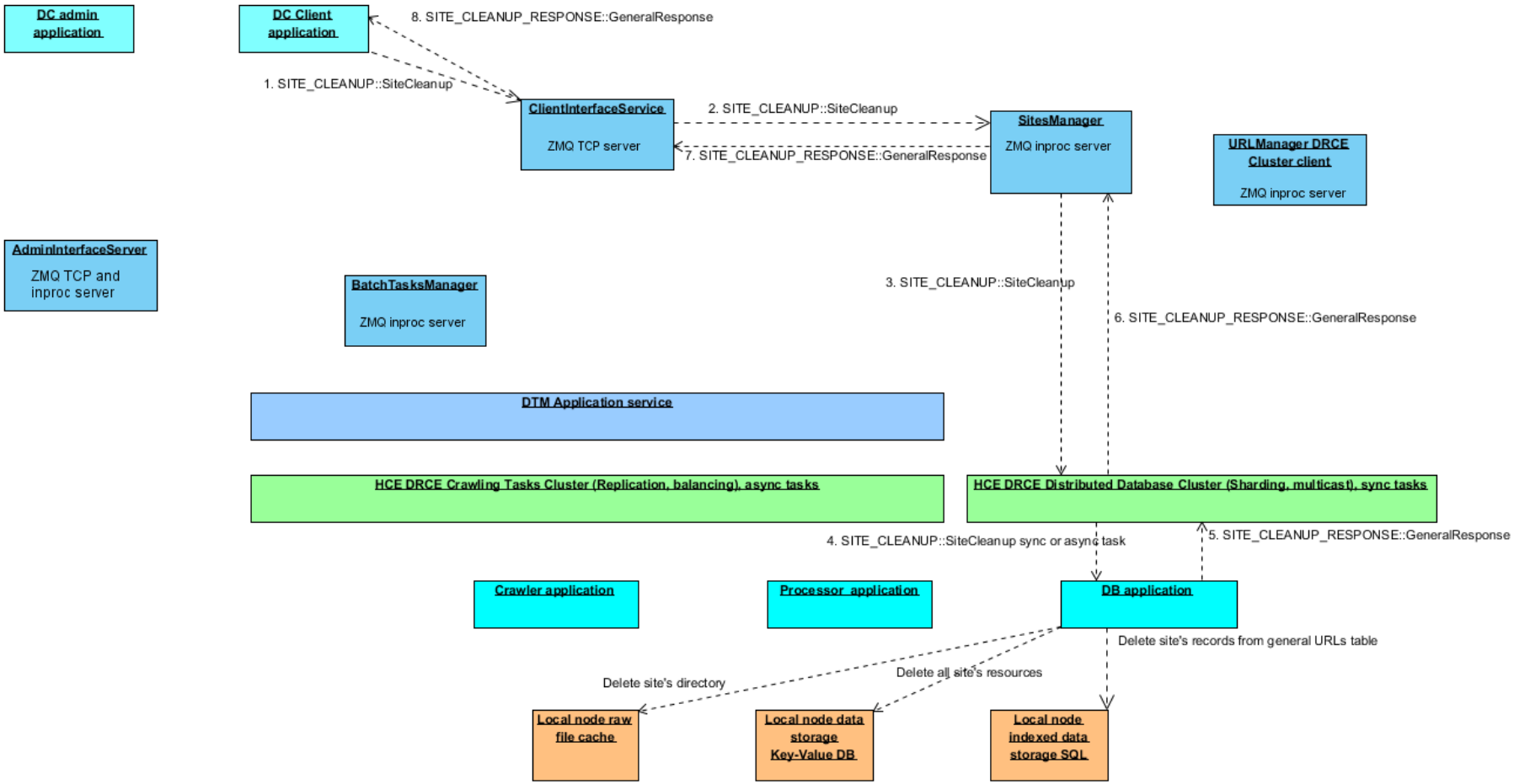
The processing

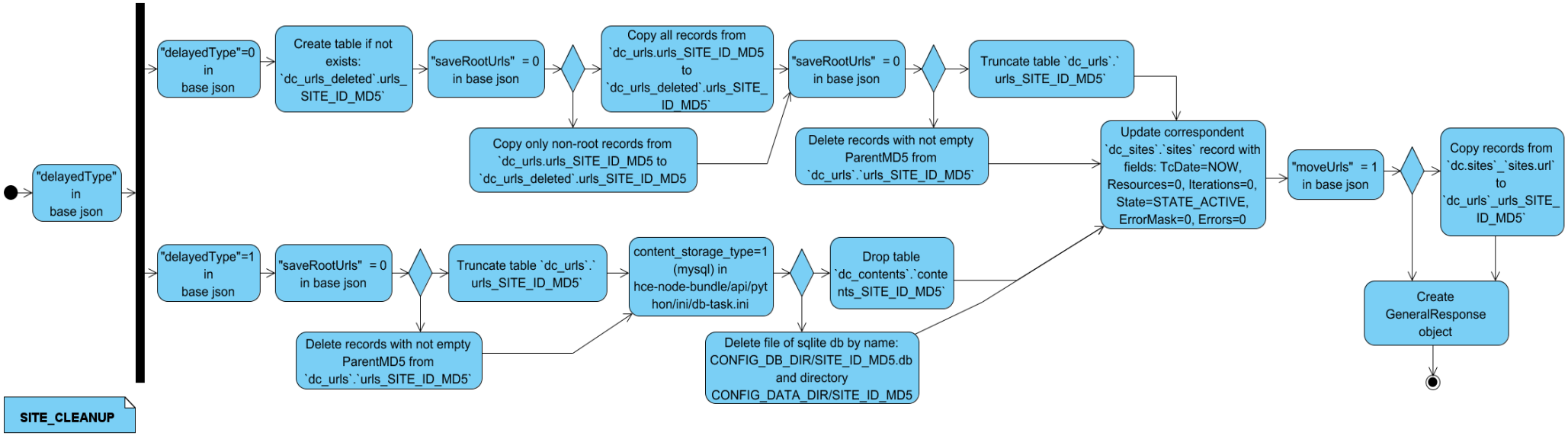
The processing is periodic process that creates the URLFetch object and sets it as DTM task to execute. The execution has three steps: 1) execution of the URLFetch operation, 2) convert the URLs list to the Batch object and 3) execution of the processing operation. The result of the processing depends on the processing method or algorithm, typically it is a scraping. The processed resulted content are stored depends on the method, for the scraping it is a local storage. The URLs objects inside a DC service that are processed changed status to PROCESSED(7).


DC Admin application
DC Client application
AdminInterfaceServer
ZMQ TCP and Inproc server
ClientInterfaceServer
ZMQ TCP server
BatchTasksManagerRealTime
ZMQ inproc server
BatchTasksManagerProcess
ZMQ inproc server
BatchTasksManager
ZMQ inproc server
SitesManager
ZMQ inproc server
A1. ::NewTask
A2. ::GeneralResponse
C1. Get task results with delete results
C2. Task state response
E1. Delete task
DTM Application Service
D1. Check DRCE task state request
B1. Set new task DRCE request
HCE-NODE DRCE Crawling cluster (Replication, resources-usage balancing, asynchronous tasks, n-cluster)
HCE-NODE DRCE DB cluster (Sharding, multicast, synchronous tasks, m-cluster)
D2. Check task state
B2. ::Batch task app run
Crawler Application
DB Application
B3. Select URL(s), Create and return URL object(s)
Local node indexed data storage SQL
B4. List of URL
Update state
Convertor Application
Local node data storage Key-Value DB
B6. Insert content
B5. Batch
Processor Application
Get raw data file
Local node raw file cache
B7. Batch
PostProcessor Application
F1. Batch
F2. Batch
Social Module Application (Collect social networks data)
B8. Batch
ContentUpdater Application
DRCE Task

Site cleanup

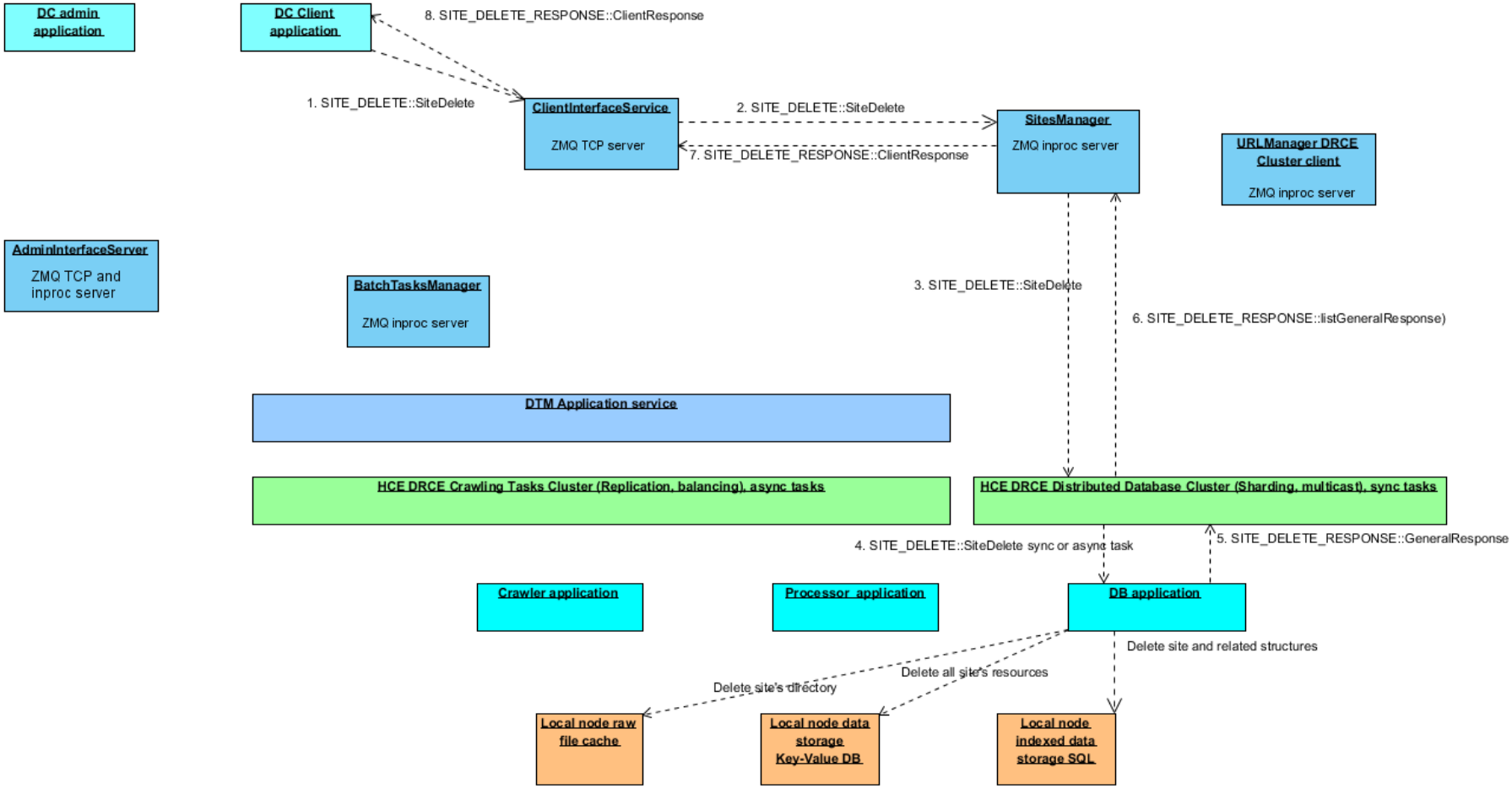
The Site object cleanup operation performs completely deletes all related URLs objects and contents and free space. Can be performed with delete of root URLs and without. If uses delete method that inherited from default.

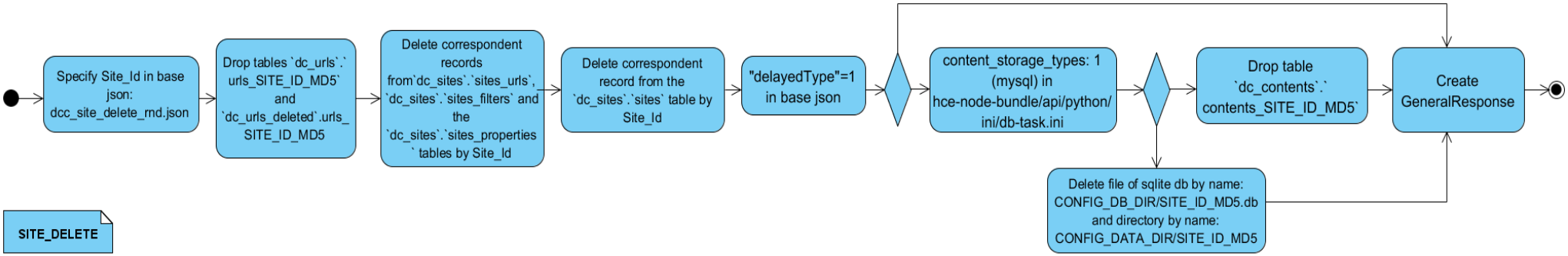



"delayedType" in base json
"delayedType"=0 in base json
Create table if not exists: `dc_urls_deleted`.urls_SITE_ID_MD5`
"saveRootUrls" = 0 in base json
Copy all records from `dc_urls.urls_SITE_ID_MD5 to `dc_urls_deleted`.urls_SITE_ID_MD5`
Copy only non-root records from `dc_urls.urls_SITE_ID_MD5 to `dc_urls_deleted`.urls_SITE_ID_MD5
"saveRootUrls" = 0 in base json
Truncate table `dc_urls`.`urls_SITE_ID_MD5`
Delete records with not empty ParentMD5 from `dc_urls`.`urls_SITE_ID_MD5`
"delayedType"=1 in base json
"saveRootUrls" = 0 in base json
Truncate table `dc_urls`.`urls_SITE_ID_MD5`
Delete records with not empty ParentMD5 from `dc_urls`.`urls_SITE_ID_MD5`
content_storage_type=1 (mysql) in hce-node-bundle/api/python/ini/db-task.ini
Drop table `dc_contents`.`contents_SITE_ID_MD5`
Delete file of sqlite db by name: CONFIG_DB_DIR/SITE_ID_MD5.db and directory CONFIG_DATA_DIR/SITE_ID_MD5
Update correspondent `dc_sites`.`sites` record with fields: TcDate=NOW, Resources=0, Iterations=0, State=STATE_ACTIVE, ErrorMask=0, Errors=0
"moveUrls" = 1 in base json
Copy records from `dc.sites`_`sites.url` to `dc_urls`_urls_SITE_ID_MD5`
Create GeneralResponse object
SITE_CLEANUP

Site delete

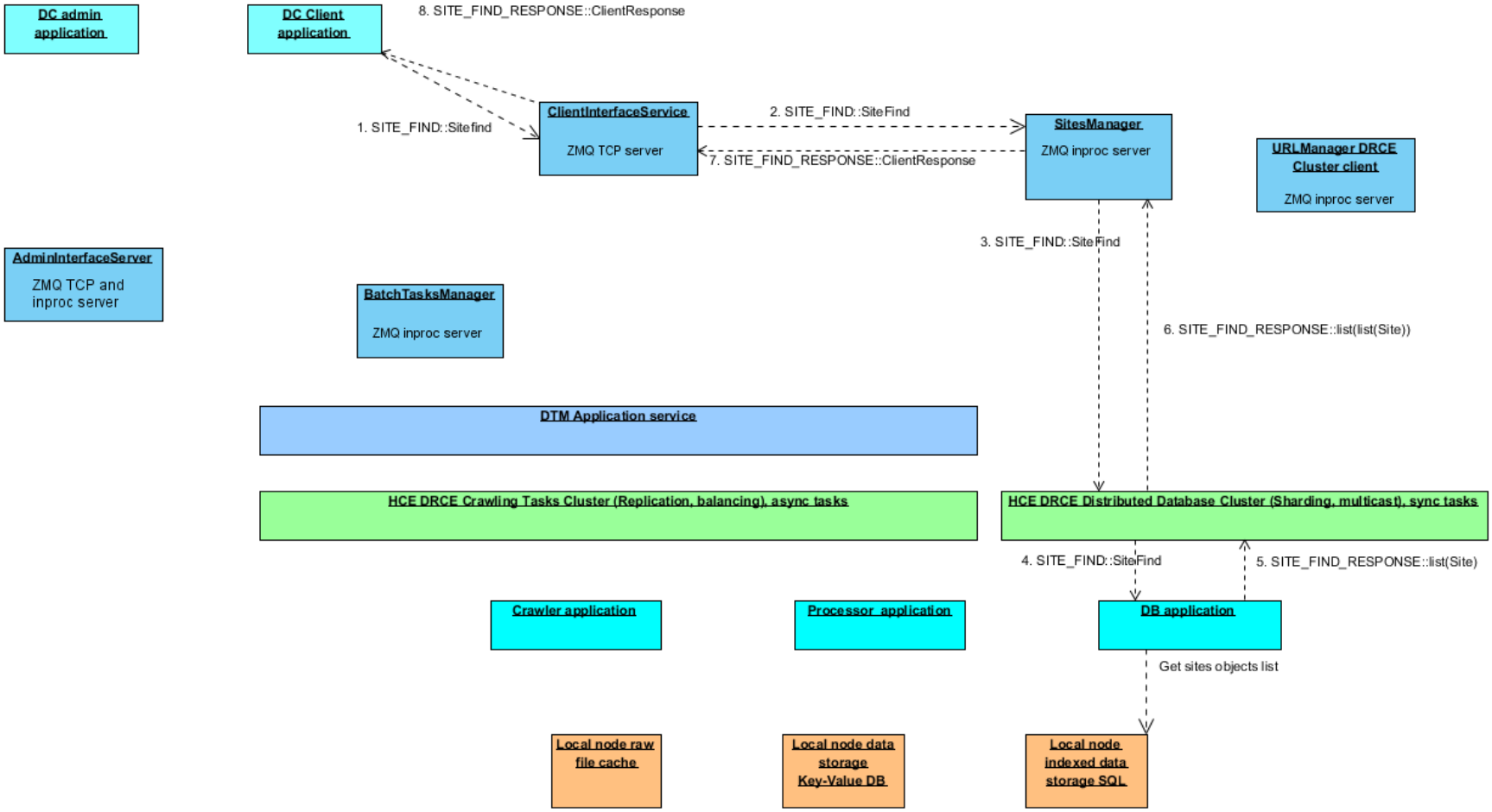
Site delete removes the Site representation from the service storages and completely cleans all related structures, delete all accumulated data and lists of items like URLs or documents.

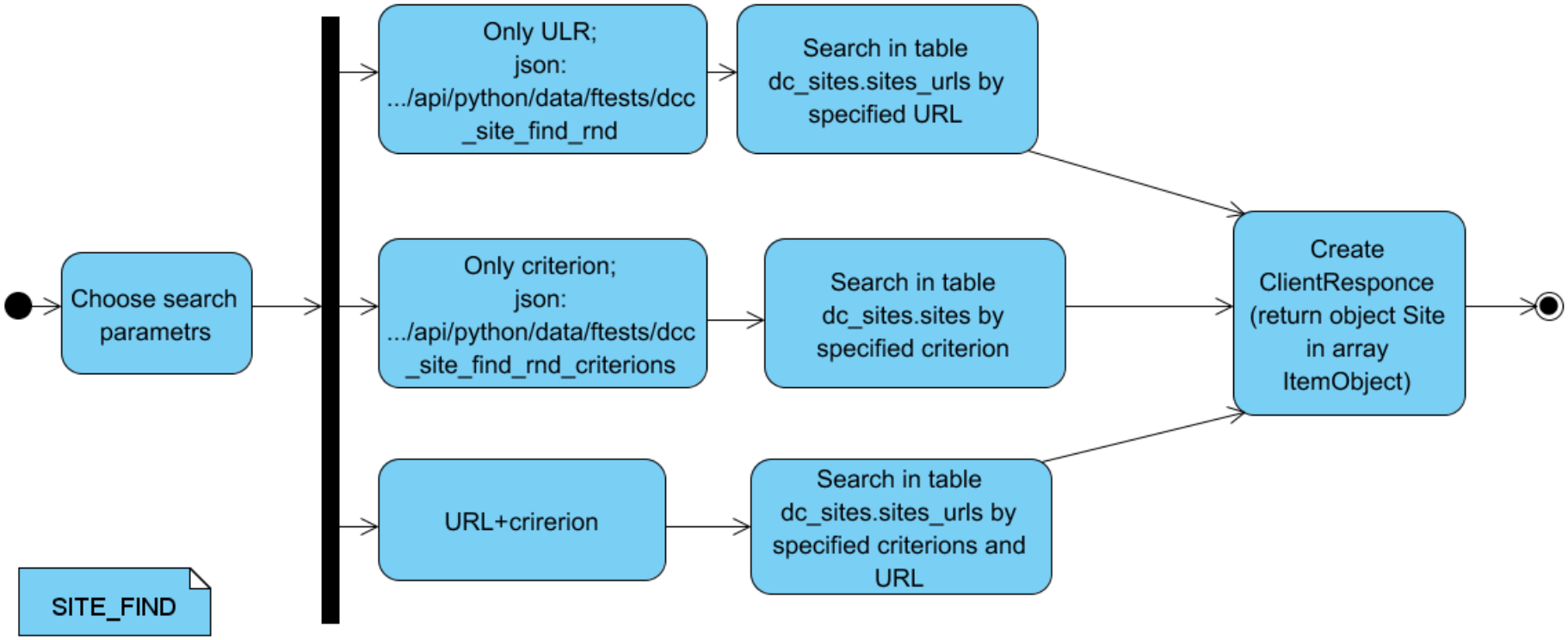



Specify Site_Id in base json: dcc_site_delete_rnd.json
Drop tables `dc_urls`.`urls_SITE_ID_MD5` and `dc_urls_deleted`.urls_SITE_ID_MD5
Delete correspondent records from`dc_sites`.`sites_urls`, `dc_sites`.`sites_filters` and the `dc_sites`.`sites_properties` tables by Site_Id
Delete correspondent record from the `dc_sites`.`sites` table by Site_Id
"delayedType"=1 in base json
content_storage_types: 1 (mysql) in hce-node-bundle/api/python/ini/db-task.ini
Drop table `dc_contents`.`contents_SITE_ID_MD5`
Create GeneralResponse
Delete file of sqlite db by name: CONFIG_DB_DIR/SITE_ID_MD5.db and directory by name: CONFIG_DATA_DIR/SITE_ID_MD5
SITE_DELETE

Site find

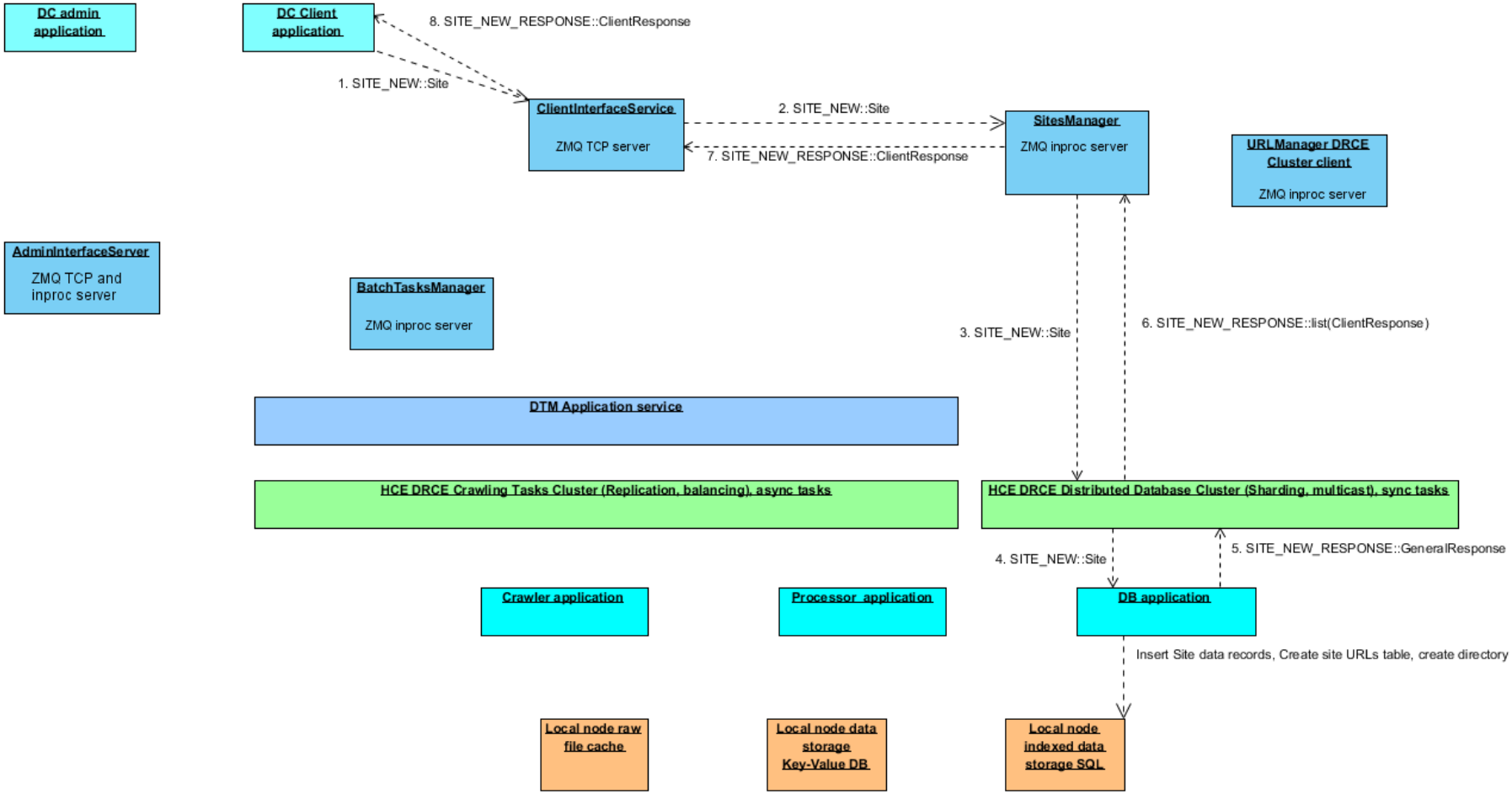
Site find performs search action with defined criterions or URL pattern. If URL pattern used – root URLs of all sites in the service compared. In results list of the Site objects returned or empty list if no one site matched.

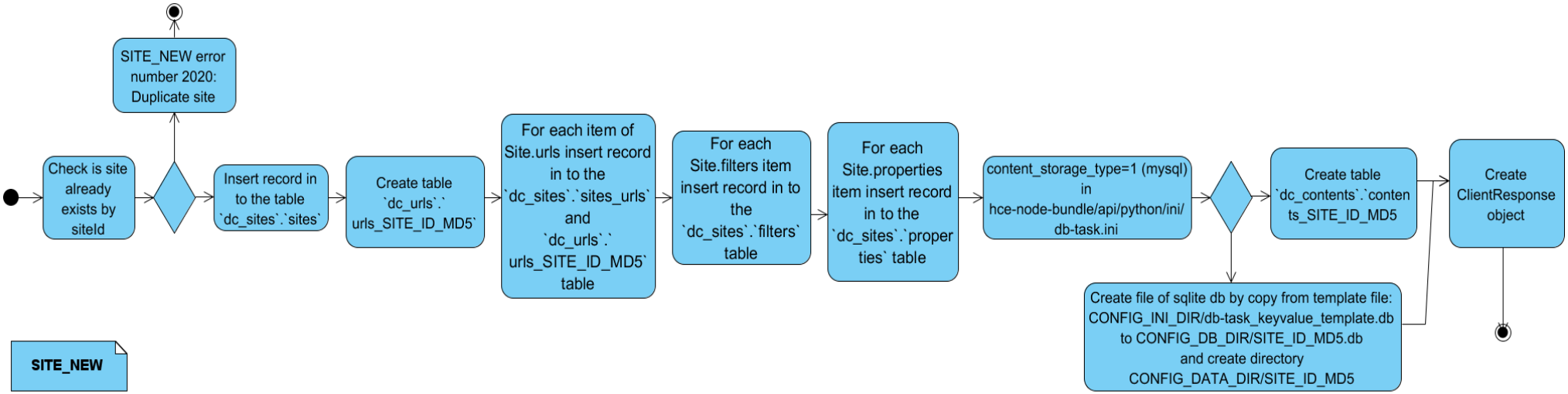



Only ULR;
json:
.../api/python/data/ftests/dcc
_site_find_rnd
Search in table
dc_sites.sites_urls by
specified URL
Choose search
parametrs
Only criterion;
json:
.../api/python/data/ftests/dcc
_site_find_rnd_criterions
Search in table
dc_sites.sites by
specified criterion
Create
ClientResponce
(return object Site
in array
ItemObject)
URL+crirerion
Search in table
dc_sites.sites_urls by
specified criterions and
URL
SITE_FIND

Site new

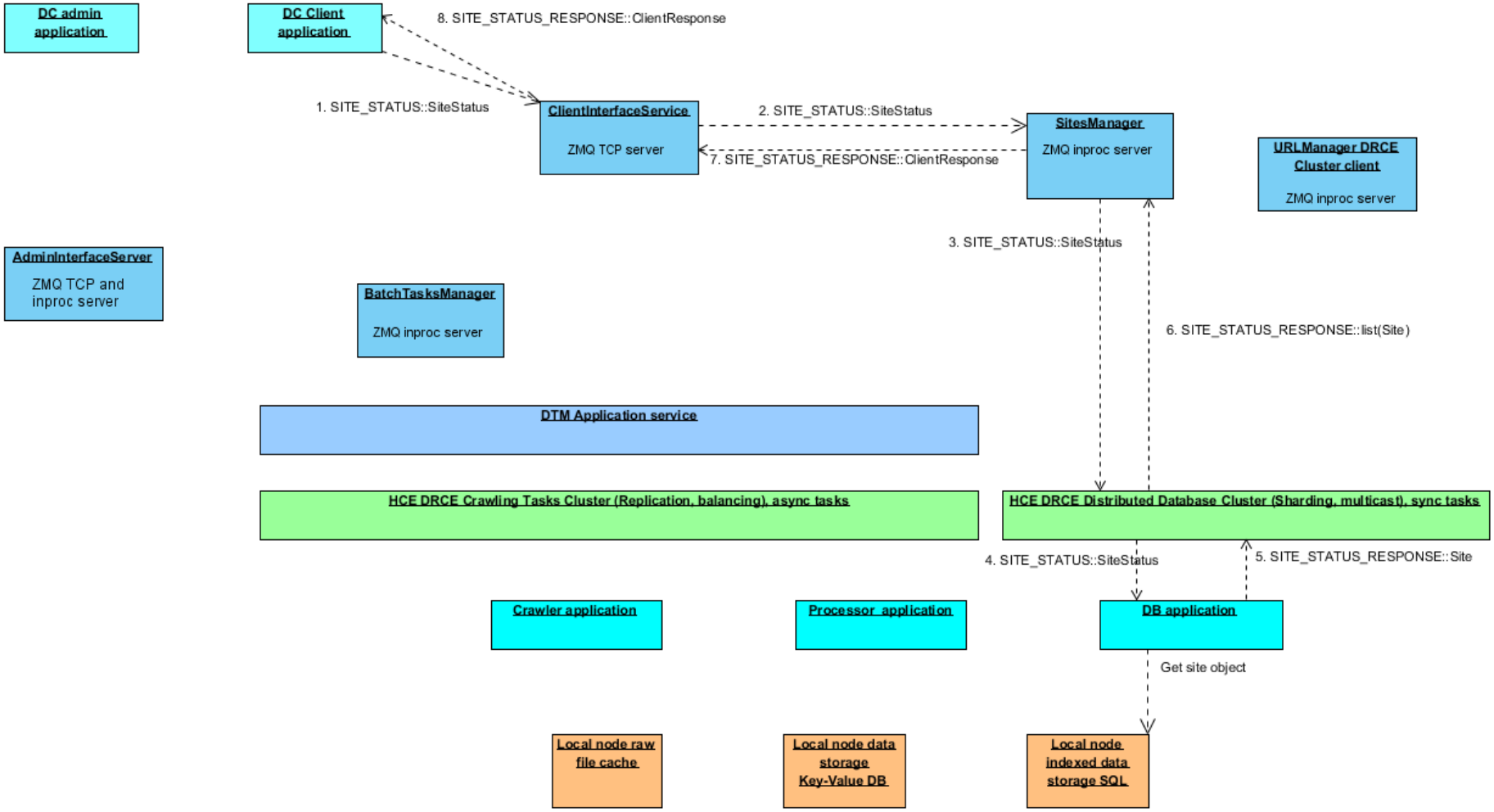
Creates new Site object representation inside the service and initializes all related structures and lists of related objects. Site can to start to take a part in the regular crawling or another periodic processes depends on initial values of its firlds.

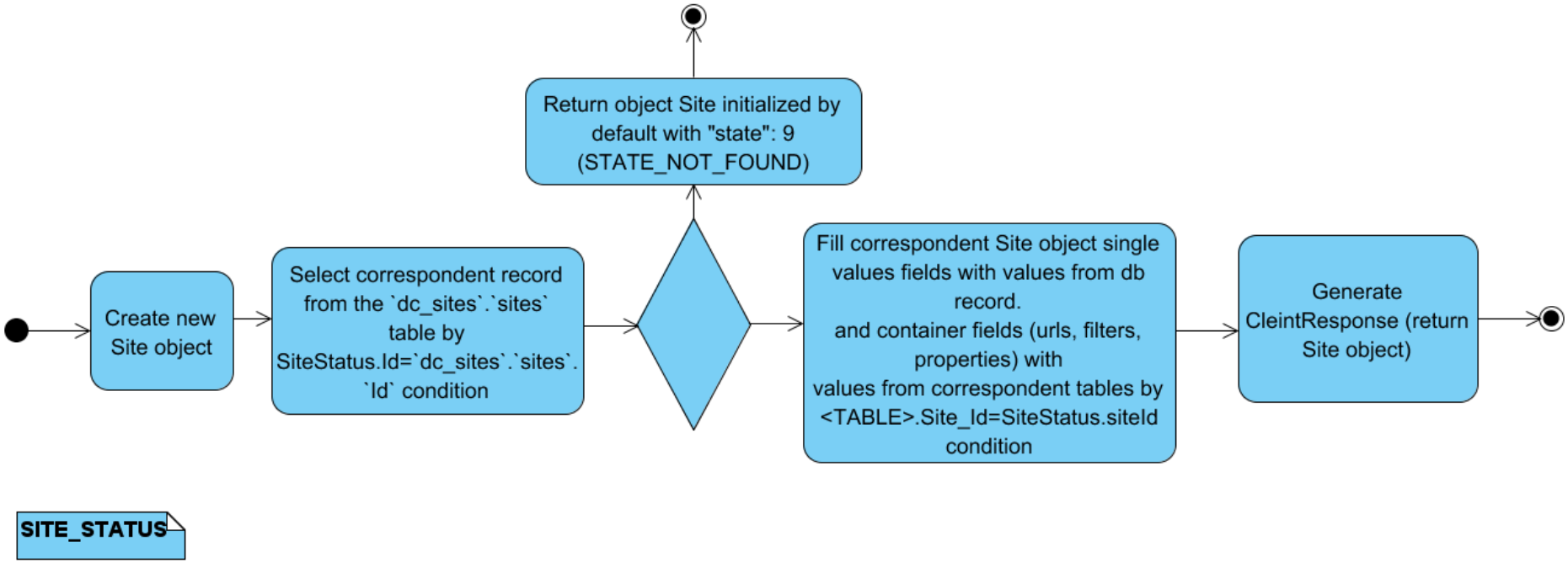



SITE_NEW error number 2020: Duplicate site
Check is site already exists by siteId
Insert record in to the table `dc_sites`.`sites`
Create table `dc_urls`.` urls_SITE_ID_MD5`
For each item of Site.urls insert record in to the `dc_sites`.`sites_urls` and `dc_urls`.` urls_SITE_ID_MD5` table
For each Site.filters item insert record in to the `dc_sites`.`filters` table
For each Site.properties item insert record in to the `dc_sites`.`properties` table
content_storage_type=1 (mysql) in hce-node-bundle/api/python/ini/db-task.ini
Create table `dc_contents`.`contents_SITE_ID_MD5
Create ClientResponse object
Create file of sqlite db by copy from template file: CONFIG_INI_DIR/db-task_keyvalue_template.db to CONFIG_DB_DIR/SITE_ID_MD5.db and create directory CONFIG_DATA_DIR/SITE_ID_MD5
SITE_NEW

Site status

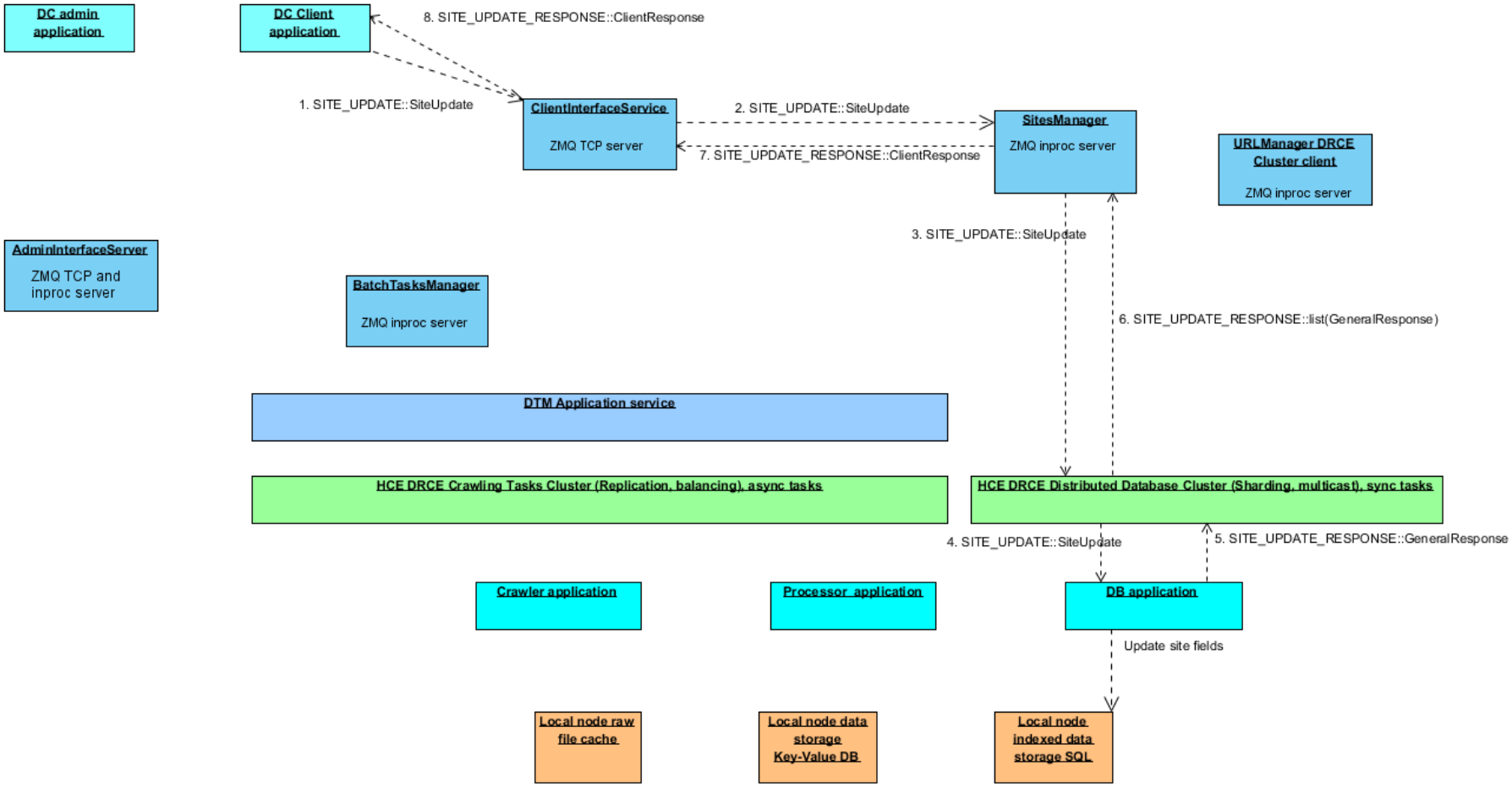
This operation performs simple check of presence of the Site object in the system by the site Id and returns it. If site not present – error code and error message set. Operation can be used to get the Site object fast way without involving the search process.

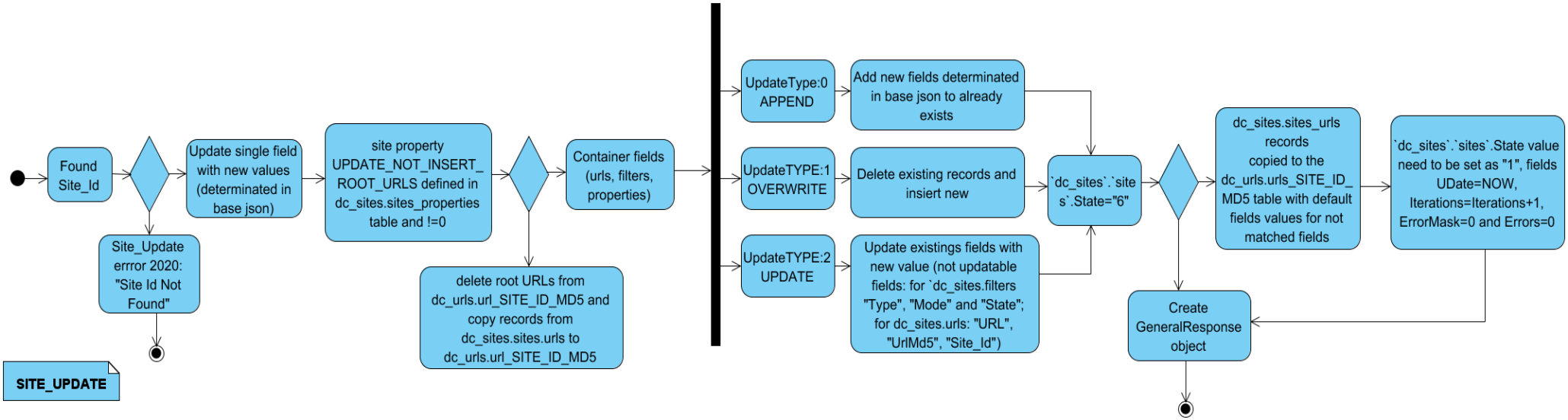



Return object Site initialized by default with "state": 9 (STATE_NOT_FOUND)
Create new Site object
Select correspondent record from the `dc_sites`.`sites` table by SiteStatus.Id=`dc_sites`.`sites`.`Id` condition
Fill correspondent Site object single values fields with values from db record.
and container fields (urls, filters, properties) with
values from correspondent tables by <TABLE>.Site_Id=SiteStatus.siteId condition
Generate CleintResponse (return Site object)
SITE_STATUS

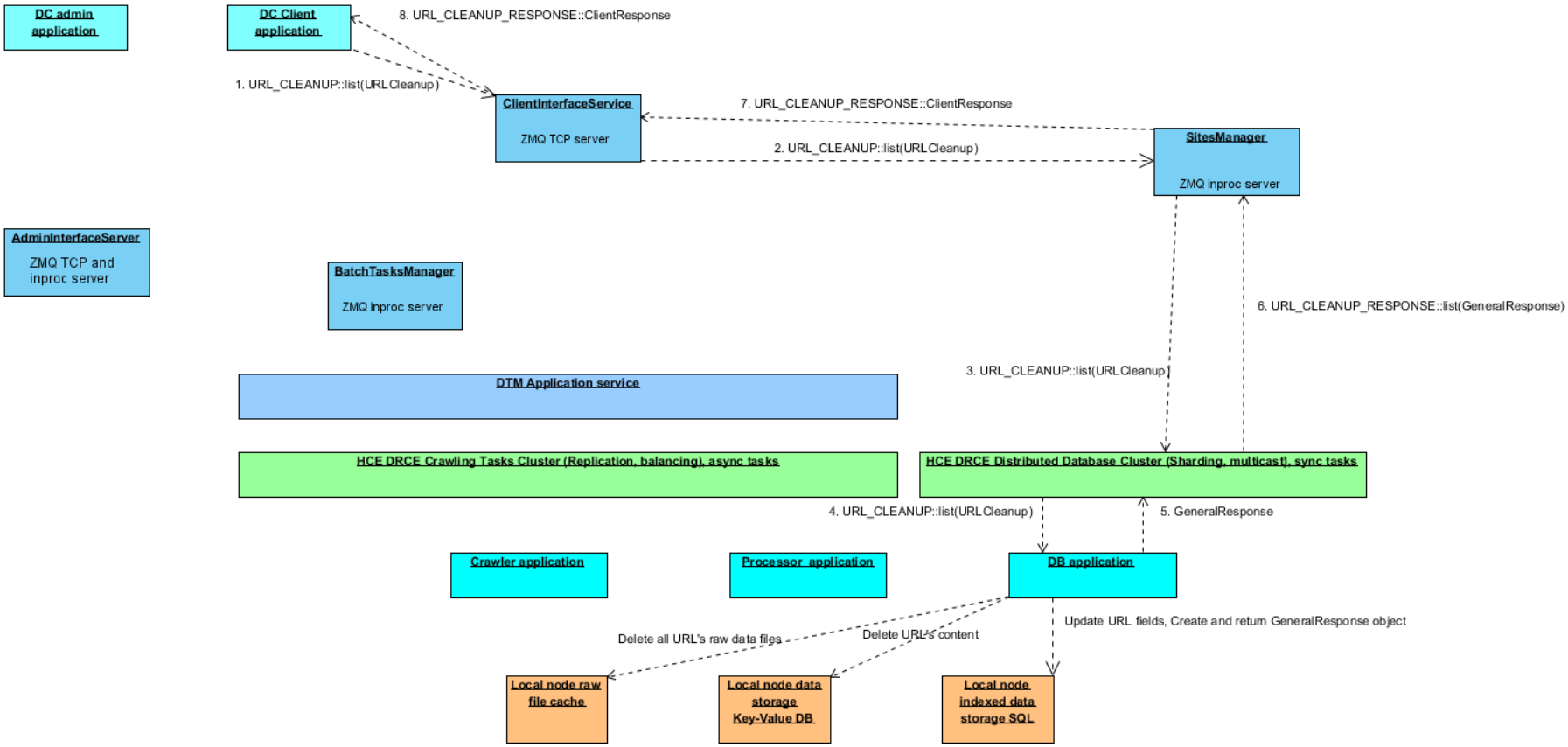
Site update

This operation updates the Site object representation inside the service and/or related data structures. For multi-item structures like lists of objects like root URLs, filters or templates three different actions possible – append, overwrite and update. Note that operation is complex and can be done partially in case of some database or another kind errors.

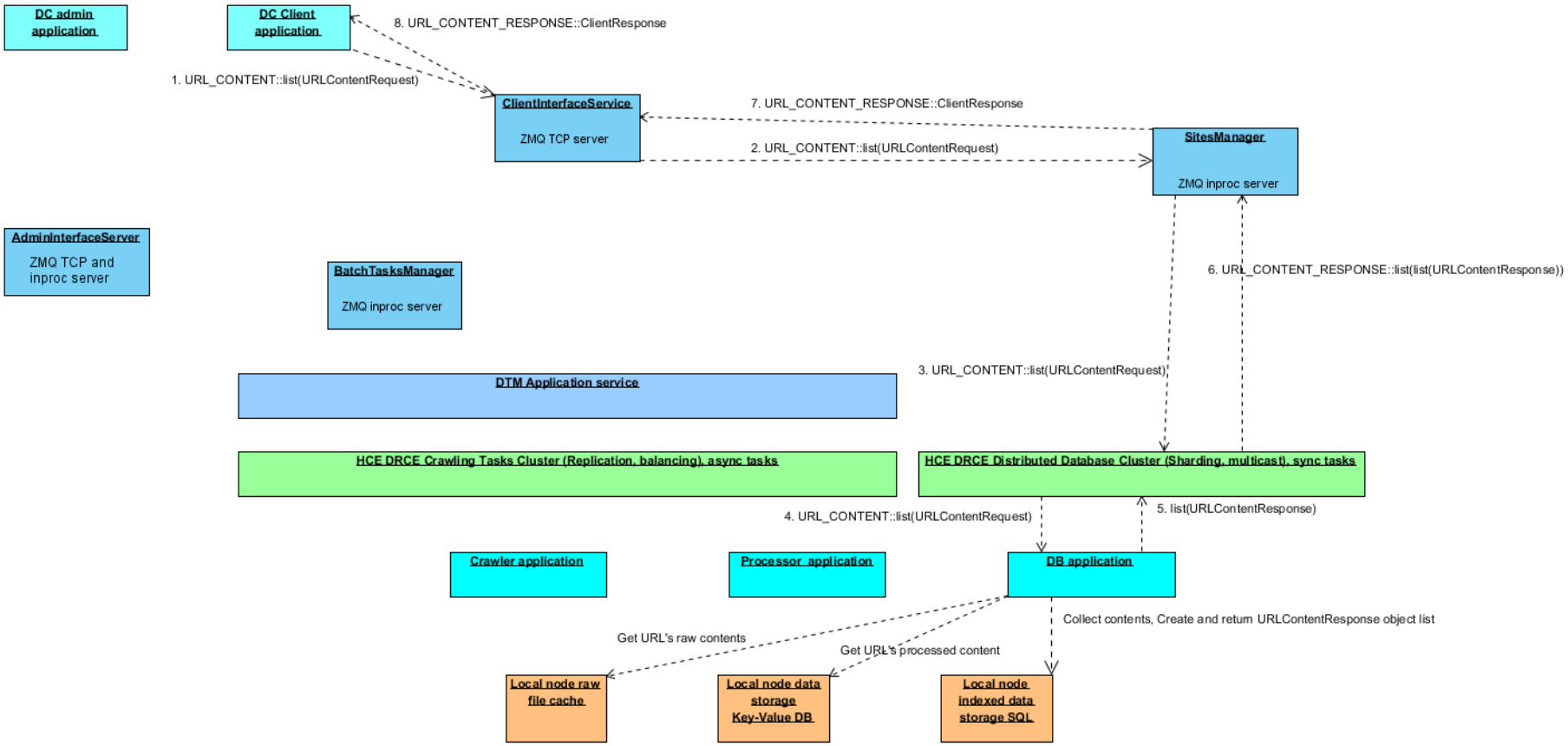



Found Site_Id
Site_Update errror 2020: "Site Id Not Found"
Update single field with new values (determinated in base json)
site property UPDATE_NOT_INSERT_ROOT_URLS defined in dc_sites.sites_properties table and !=0
delete root URLs from dc_urls.url_SITE_ID_MD5 and copy records from dc_sites.sites.urls to dc_urls.url_SITE_ID_MD5
Container fields (urls, filters, properties)
UpdateType:0 APPEND
Add new fields determinated in base json to already exists
UpdateTYPE:1 OVERWRITE
Delete existing records and insiert new
UpdateTYPE:2 UPDATE
Update existings fields with new value (not updatable fields: for `dc_sites.filters "Type", "Mode" and "State"; for dc_sites.urls: "URL", "UrlMd5", "Site_Id")
`dc_sites`.`sites`.State="6"
dc_sites.sites_urls records copied to the dc_urls.urls_SITE_ID_MD5 table with default fields values for not matched fields
`dc_sites`.`sites`.State value need to be set as "1", fields UDate=NOW, Iterations=Iterations+1, ErrorMask=0 and Errors=0
Create GeneralResponse object
SITE_UPDATE

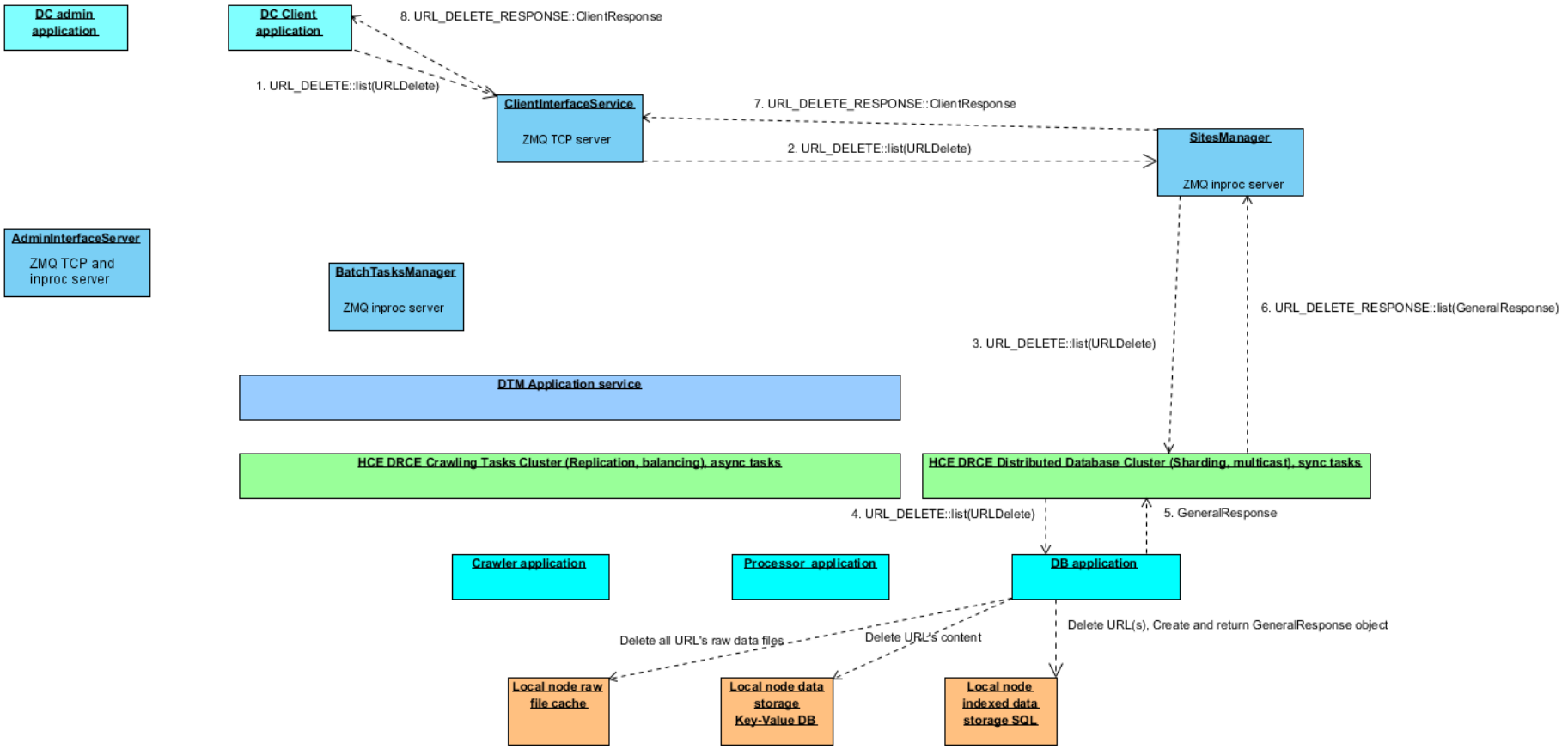
URL cleanup


DC admin application
DC Client application
8. URL_CLEANUP_RESPONSE::ClientResponse
1. URL_CLEANUP::list(URLCleanup)
ClientInterfaceService
ZMQ TCP server
7. URL_CLEANUP_RESPONSE::ClientResponse
2. URL_CLEANUP::list(URLCleanup)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_CLEANUP_RESPONSE::list(GeneralResponse)
3. URL_CLEANUP::list(URLCleanup)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_CLEANUP::list(URLCleanup)
5. GeneralResponse
Crawler application
Processor application
DB application
Update URL fields, Create and return GeneralResponse object
Delete all URL's raw data files
Delete URL's content
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

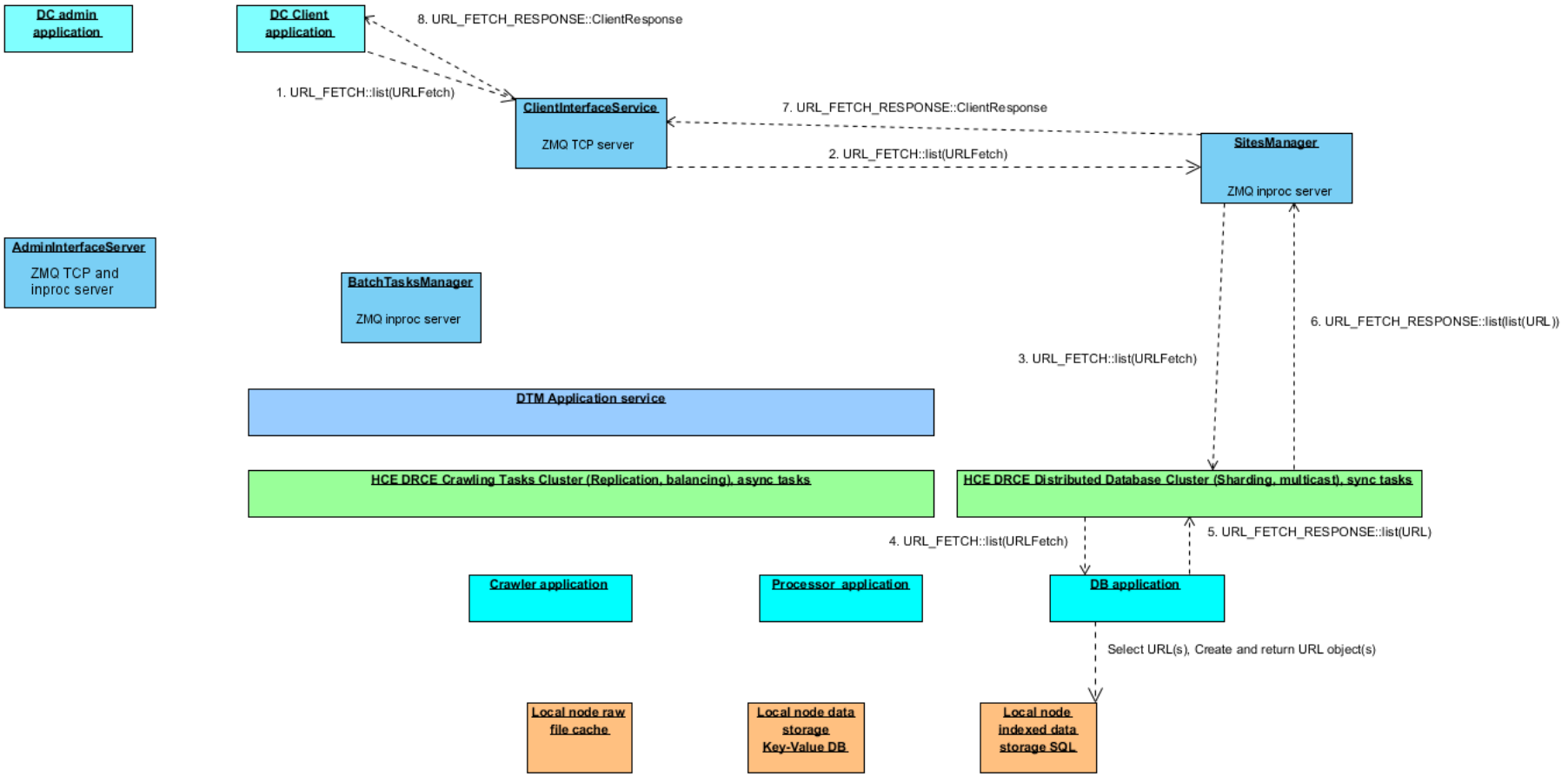
URL content


DC admin application
DC Client application
8. URL_CONTENT_RESPONSE::ClientResponse
1. URL_CONTENT::list(URLContentRequest)
ClientInterfaceService
ZMQ TCP server
7. URL_CONTENT_RESPONSE::ClientResponse
2. URL_CONTENT::list(URLContentRequest)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_CONTENT_RESPONSE::list(list(URLContentResponse))
3. URL_CONTENT::list(URLContentRequest)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_CONTENT::list(URLContentRequest)
5. list(URLContentResponse)
Crawler application
Processor application
DB application
Collect contents, Create and return URLContentResponse object list
Get URL's raw contents
Get URL's processed content
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

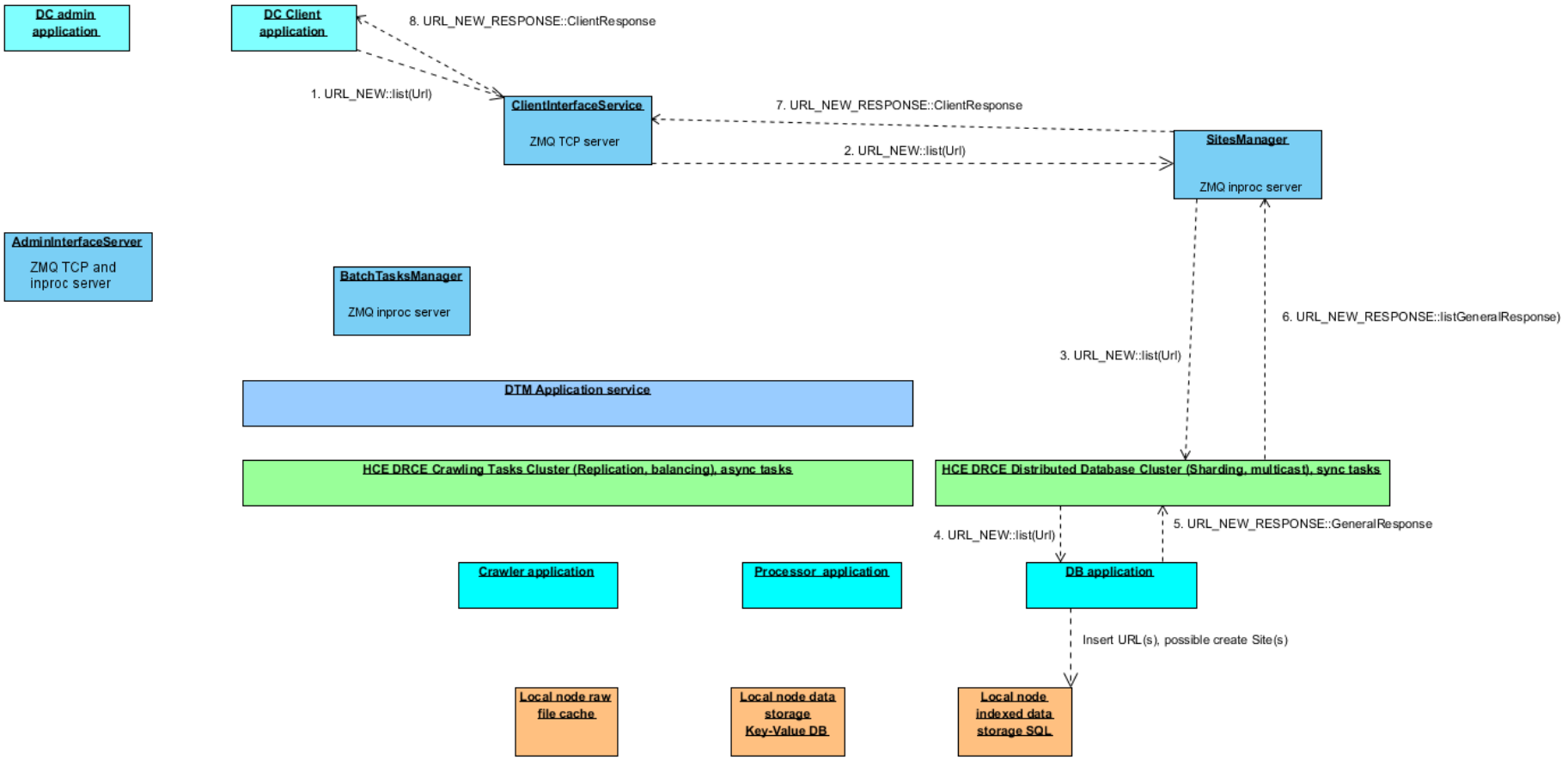
URL delete


DC admin application
DC Client application
8. URL_DELETE_RESPONSE::ClientResponse
1. URL_DELETE::list(URLDelete)
ClientInterfaceService
ZMQ TCP server
7. URL_DELETE_RESPONSE::ClientResponse
2. URL_DELETE::list(URLDelete)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_DELETE_RESPONSE::list(GeneralResponse)
3. URL_DELETE::list(URLDelete)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_DELETE::list(URLDelete)
5. GeneralResponse
Crawler application
Processor application
DB application
Delete URL(s), Create and return GeneralResponse object
Delete all URL's raw data files
Delete URL's content
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

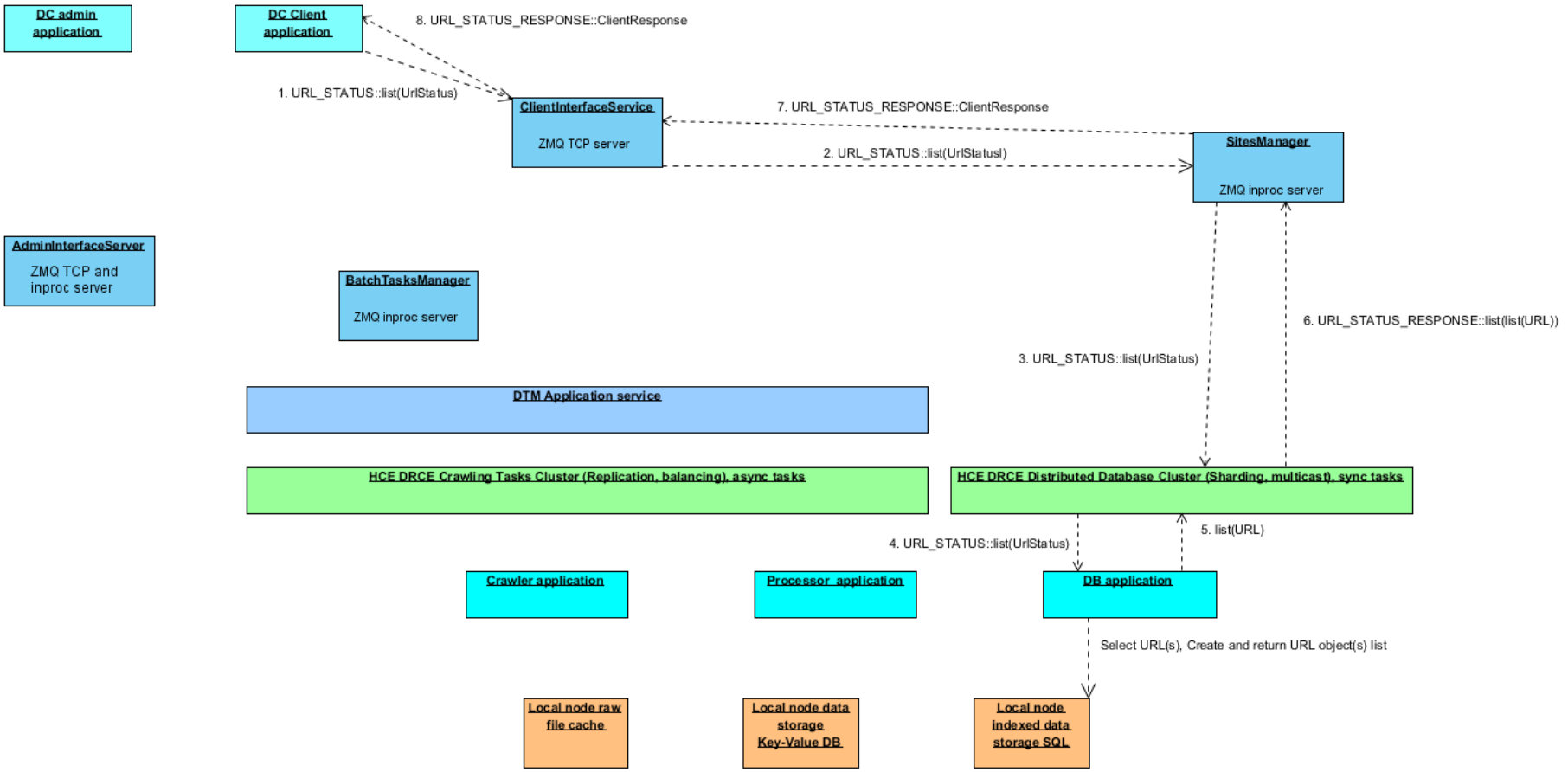
URL fetch


DC admin application
DC Client application
8. URL_FETCH_RESPONSE::ClientResponse
1. URL_FETCH::list(URLFetch)
ClientInterfaceService
ZMQ TCP server
7. URL_FETCH_RESPONSE::ClientResponse
2. URL_FETCH::list(URLFetch)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_FETCH_RESPONSE::list(list(URL))
3. URL_FETCH::list(URLFetch)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_FETCH::list(URLFetch)
5. URL_FETCH_RESPONSE::list(URL)
Crawler application
Processor application
DB application
Select URL(s), Create and return URL object(s)
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

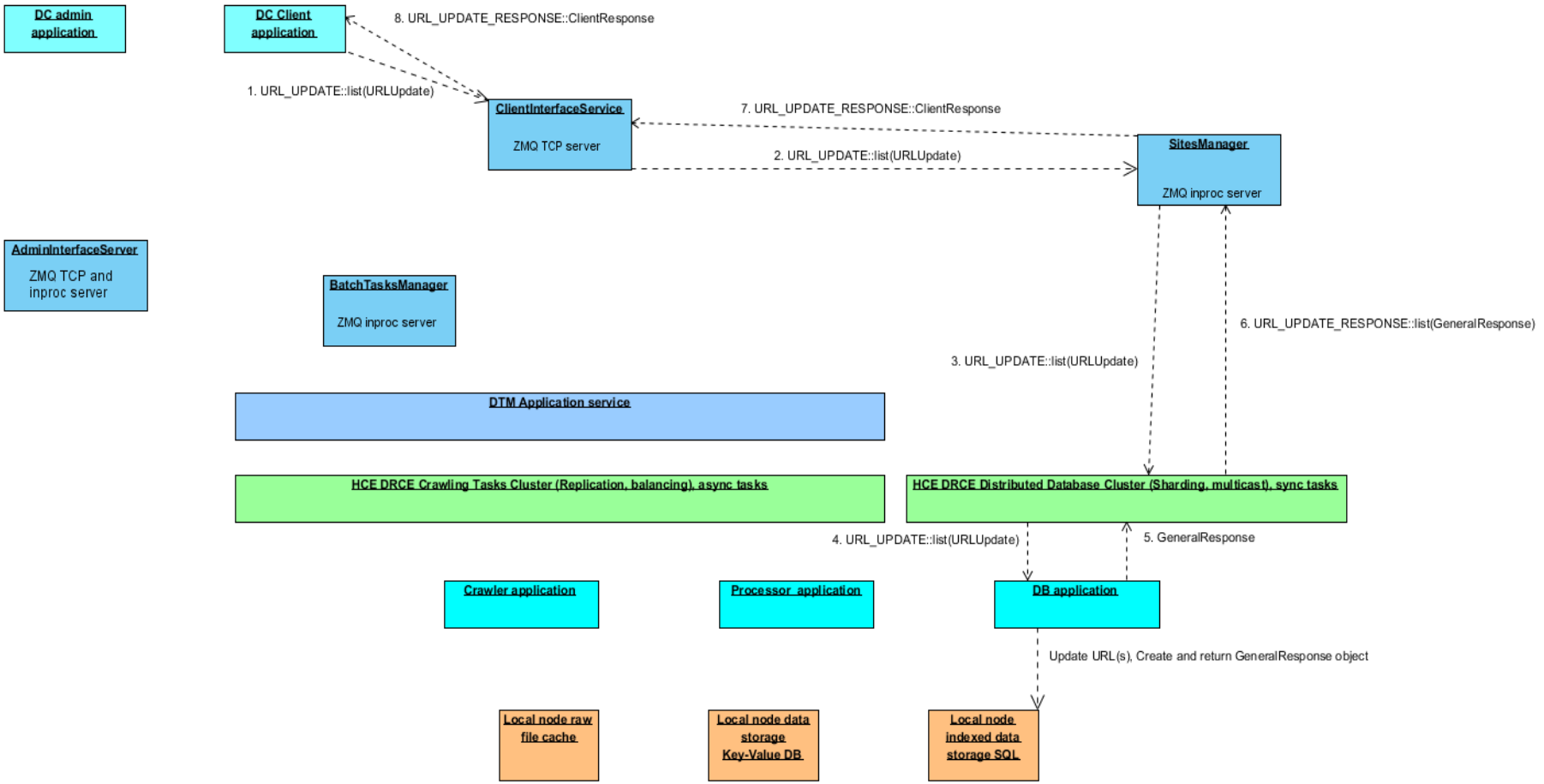
URL new


DC admin application
DC Client application
8. URL_NEW_RESPONSE::ClientResponse
1. URL_NEW::list(Url)
ClientInterfaceService
ZMQ TCP server
7. URL_NEW_RESPONSE::ClientResponse
2. URL_NEW::list(Url)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_NEW_RESPONSE::listGeneralResponse)
3. URL_NEW::list(Url)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_NEW::list(Url)
5. URL_NEW_RESPONSE::GeneralResponse
Crawler application
Processor application
DB application
Insert URL(s), possible create Site(s)
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

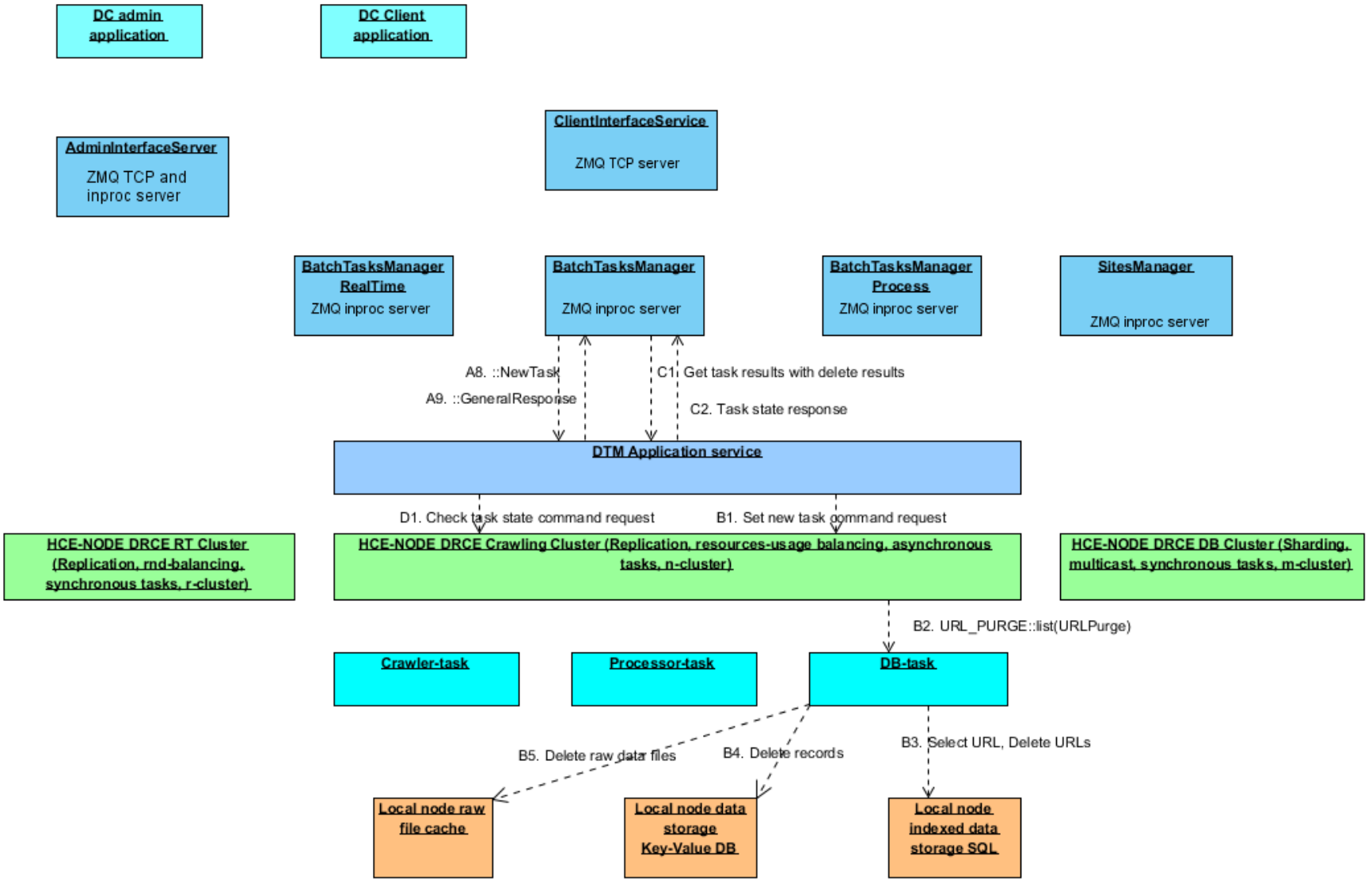
URL status


DC admin application
DC Client application
8. URL_STATUS_RESPONSE::ClientResponse
1. URL_STATUS::list(UrlStatus)
ClientInterfaceService
ZMQ TCP server
7. URL_STATUS_RESPONSE::ClientResponse
2. URL_STATUS::list(UrlStatusl)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_STATUS_RESPONSE::list(list(URL))
3. URL_STATUS::list(UrlStatus)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
5. list(URL)
4. URL_STATUS::list(UrlStatus)
Crawler application
Processor application
DB application
Select URL(s), Create and return URL object(s) list
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

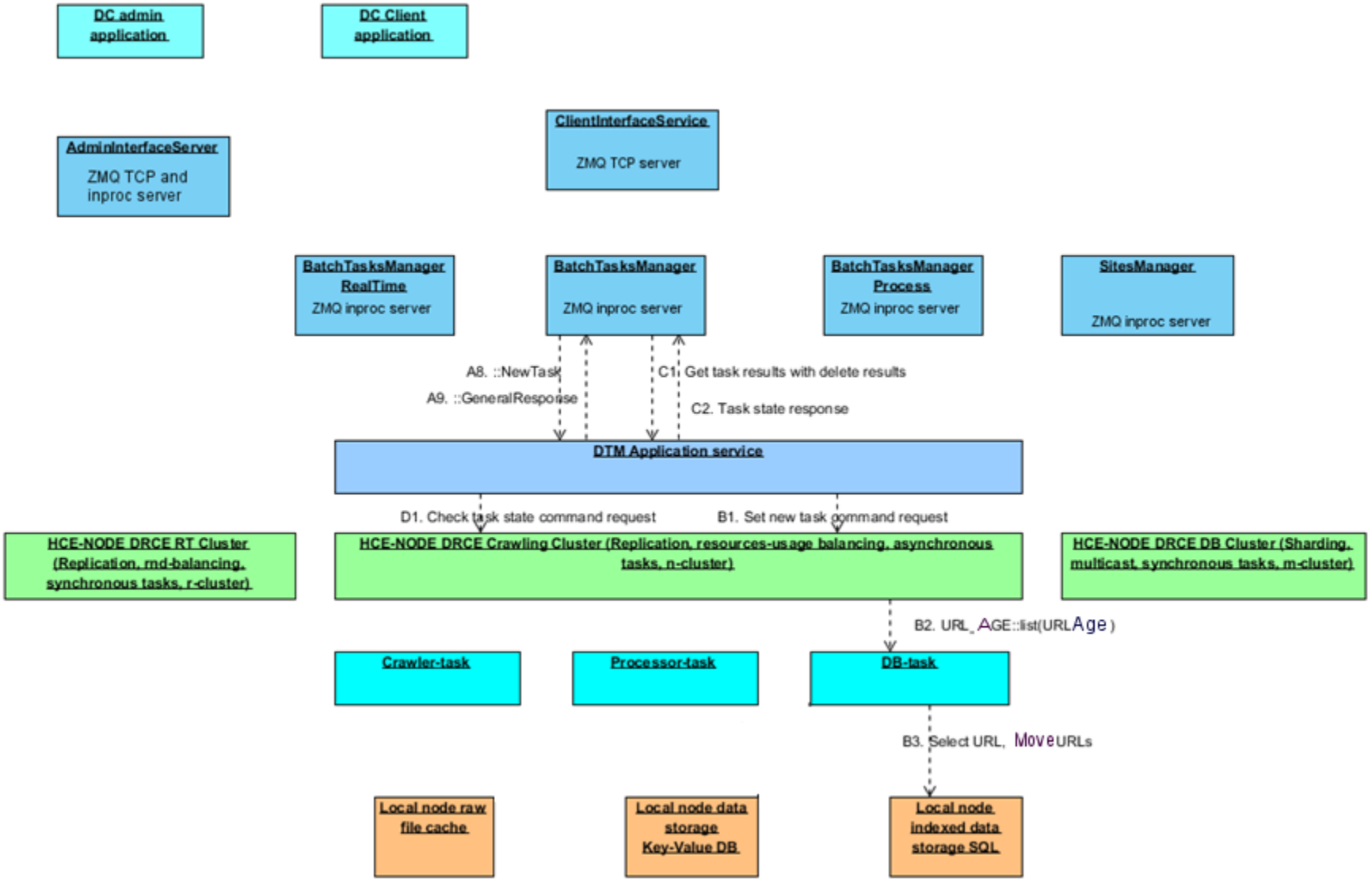
URL update


DC admin application
DC Client application
8. URL_UPDATE_RESPONSE::ClientResponse
1. URL_UPDATE::list(URLUpdate)
ClientInterfaceService
ZMQ TCP server
7. URL_UPDATE_RESPONSE::ClientResponse
2. URL_UPDATE::list(URLUpdate)
SitesManager
ZMQ inproc server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager
ZMQ inproc server
6. URL_UPDATE_RESPONSE::list(GeneralResponse)
3. URL_UPDATE::list(URLUpdate)
DTM Application service
HCE DRCE Crawling Tasks Cluster (Replication, balancing), async tasks
HCE DRCE Distributed Database Cluster (Sharding, multicast), sync tasks
4. URL_UPDATE::list(URLUpdate)
5. GeneralResponse
Crawler application
Processor application
DB application
Update URL(s), Create and return GeneralResponse object
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

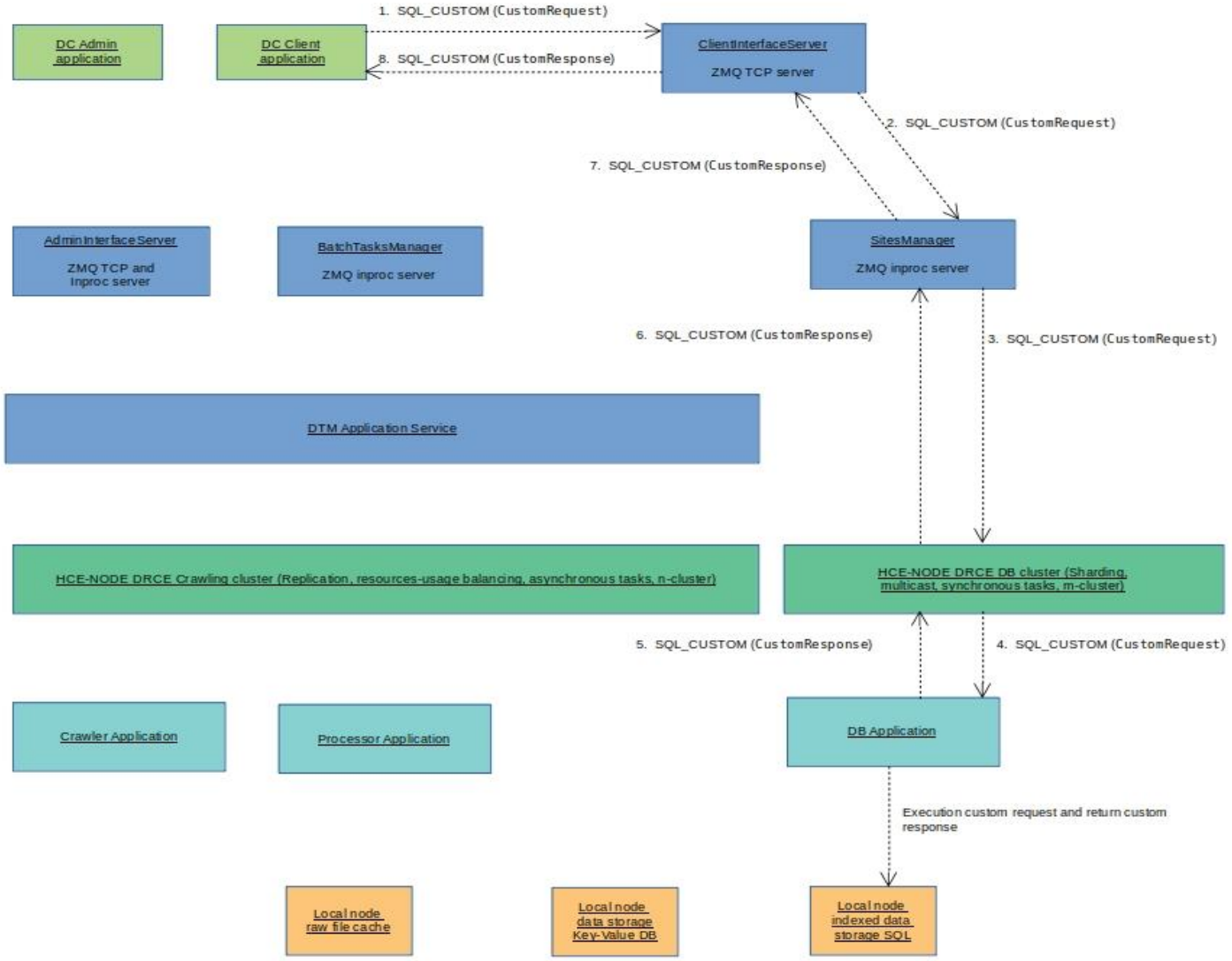
URL purge


DC admin application
DC Client application
ClientInterfaceService
ZMQ TCP server
AdminInterfaceServer
ZMQ TCP and inproc server
BatchTasksManager RealTime
ZMQ inproc server
BatchTasksManager
ZMQ inproc server
BatchTasksManager Process
ZMQ inproc server
SitesManager
ZMQ inproc server
A8. ::NewTask
A9. ::GeneralResponse
C1. Get task results with delete results
C2. Task state response
DTM Application service
D1. Check task state command request
B1. Set new task command request
HCE-NODE DRCE RT Cluster (Replication, rnd-balancing, synchronous tasks, r-cluster)
HCE-NODE DRCE Crawling Cluster (Replication, resources-usage balancing, asynchronous tasks, n-cluster)
HCE-NODE DRCE DB Cluster (Sharding, multicast, synchronous tasks, m-cluster)
B2. URL_PURGE::list(URLPurge)
Crawler-task
Processor-task
DB-task
B5. Delete raw data files
B4. Delete records
B3. Select URL, Delete URLs
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

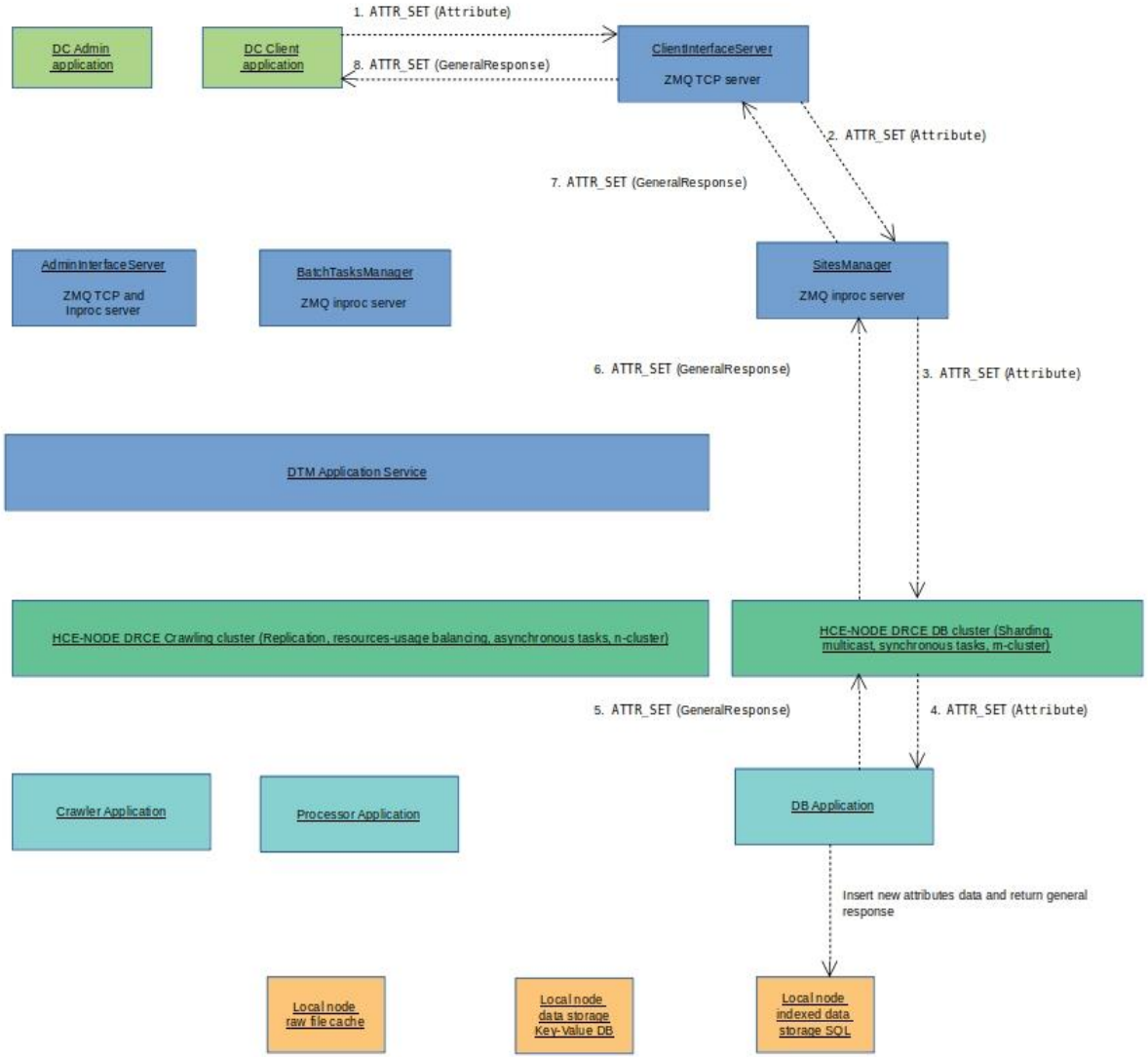
URL age


DC admin application
DC Client application
AdminInterfaceServer
ZMQ TCP and inproc server
ClientInterfaceService
ZMQ TCP server
BatchTasksManager RealTime
ZMQ inproc server
BatchTasksManager
ZMQ inproc server
BatchTasksManager Process
ZMQ inproc server
SitesManager
ZMQ inproc server
A8. ::NewTask
A9. ::GeneralResponse
C1. Get task results with delete results
C2. Task state response
DTM Application service
D1. Check task state command request
B1. Set new task command request
HCE-NODE DRCE RT Cluster (Replication, rnd-balancing, synchronous tasks, r-cluster)
HCE-NODE DRCE Crawling Cluster (Replication, resources-usage balancing, asynchronous tasks, n-cluster)
HCE-NODE DRCE DB Cluster (Sharding, multicast, synchronous tasks, m-cluster)
B2. URL_AGE::list(URLAge)
Crawler-task
Processor-task
DB-task
B3. Select URL, MoveURLs
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

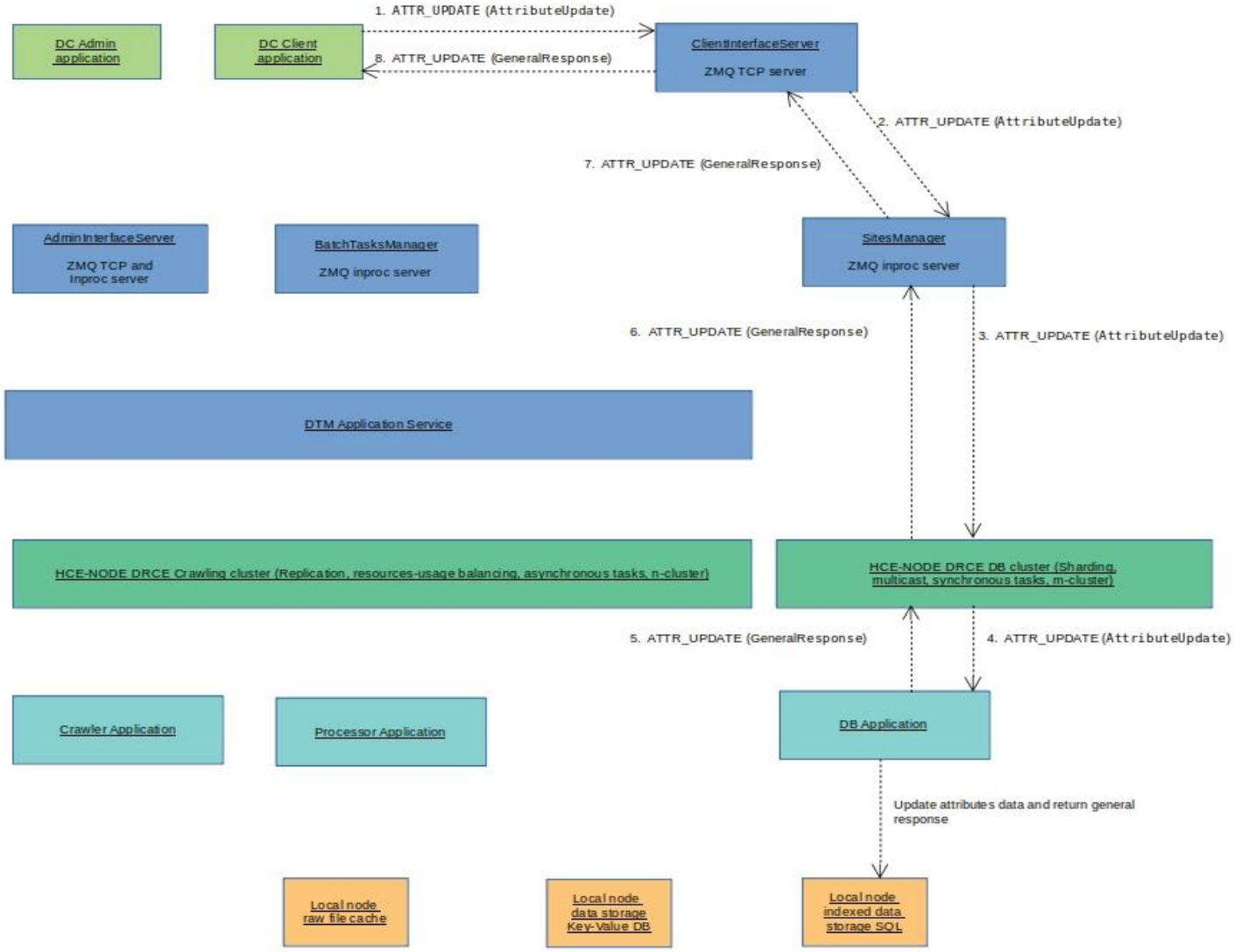
SQL Custom


1. SQL_CUSTOM (CustomRequest)
8. SQL_CUSTOM (CustomResponse)
DC Admin application
DC Client application
ClientInterfaceServer
ZMQ TCP server
2. SQL_CUSTOM (CustomRequest)
7. SQL_CUSTOM (CustomResponse)
AdminInterfaceServer
ZMQ TCP and Inproc server
BatchTasksManager
ZMQ inproc server
SitesManager
ZMQ inproc server
6. SQL_CUSTOM (CustomResponse)
3. SQL_CUSTOM (CustomRequest)
DTM Application Service
HCE-NODE DRCE Crawling cluster (Replication, resources-usage balancing, asynchronous tasks, n-cluster)
HCE-NODE DRCE DB cluster (Sharding, multicast, synchronous tasks, m-cluster)
5. SQL_CUSTOM (CustomResponse)
4. SQL_CUSTOM (CustomRequest)
Crawler Application
Processor Application
DB Application
Execution custom request and return custom response
Local node raw file cache
Local node data storage Key-Value DB
Local node indexed data storage SQL

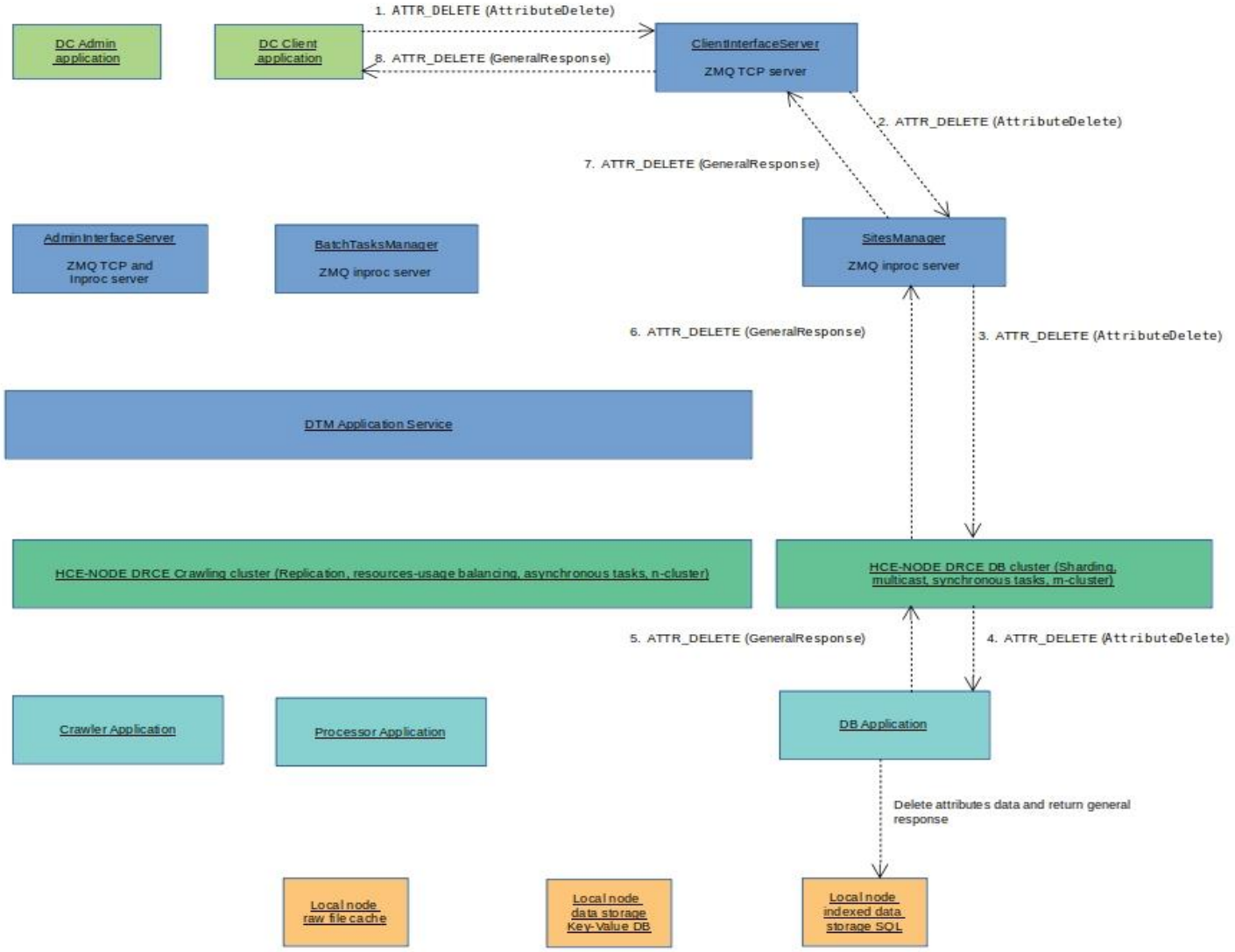
Attribute set



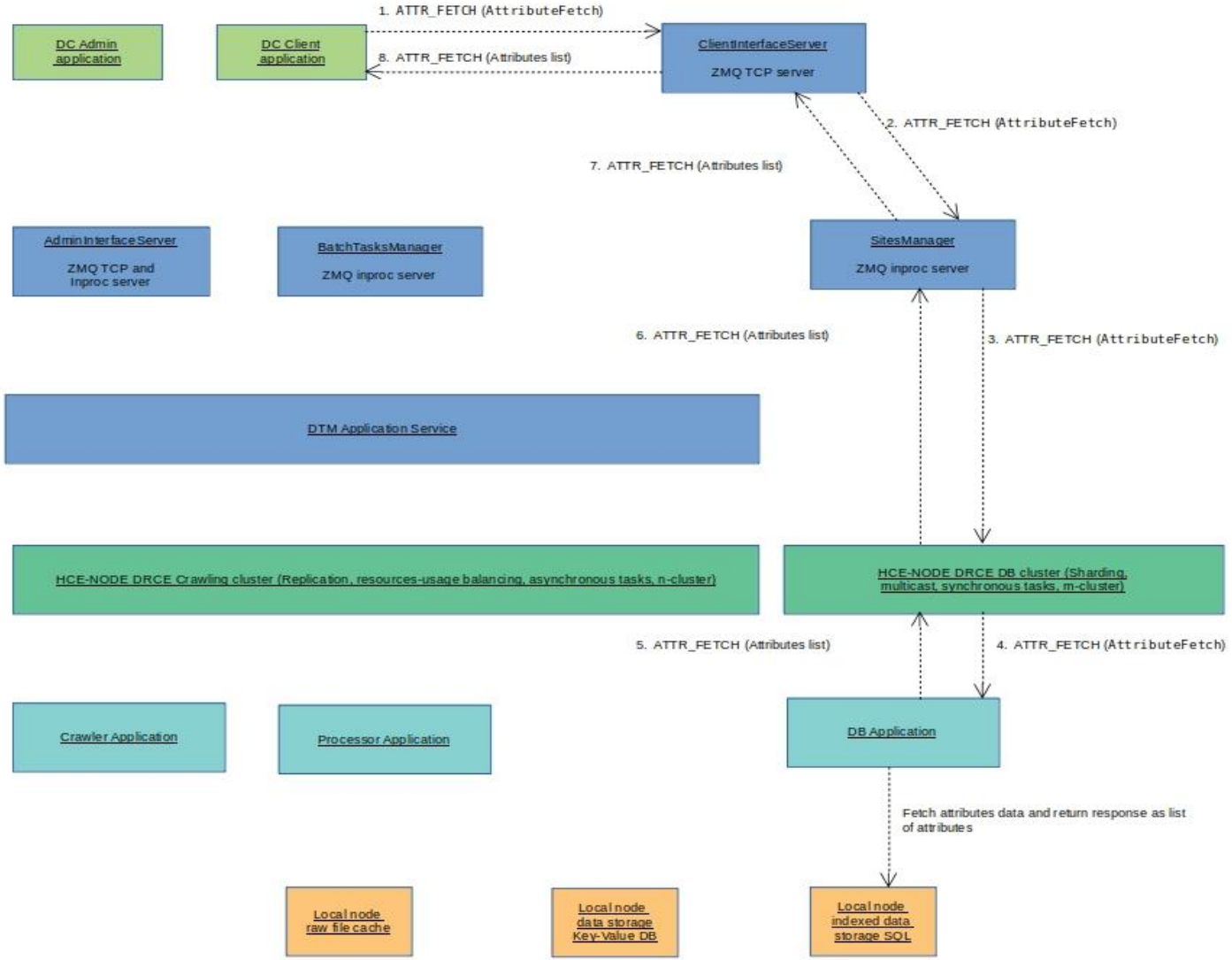
Attribute update



Attribute delete



Attribute fetch



Thread classes

AdminInterfaceServer, ClientInterfaceService, BatchTasksManager and SitesManager.

Site

The structural unit that represents crawling item from user side. Has properties applied for each URL fetched from page of this site.

Resource

The structural unit that represents results of HTTP request downloaded from site by URL. Mainly it is http resource like html page, image or any kind of MIME type content.

URL

The structural unit that represents the resource URL for the site. Used as base of main resource identifier for md5 checksum calculation.

HTTP Request

The object that used to do HTTP querying and resources downloading algorithms and operations on data nodes.

HTTP Response

The object that used to represent and to parse downloaded resources on data nodes.

Client interface service

This class accepts the client requests events from TCP connection and sends them to the proper target threading class. This class works mostly as a proxy between the target functional threaded class and clients. It connects to three target threaded classes as shown on application architecture schema.

Batch Tasks Manager

It is a main class that implements algorithms of continues crawling processing iteration, interacts with DTM application service, sets crawling tasks to execute and checks state of that tasks periodically. To start the crawling task it collects some URLs from SitesManager and creates the crawling task Batch. The crawling task batch send as main data set and input data of the main crawling application that works on target data node in the HCE DRCE Cluster.

The Crawler application

The application that acts as web-crawler, uses crawling batch as a main job order, executes HTTP requests and stores resources in the local file system cache, updates state of resources in the local SQL DB. Located and acts on data node.

This is main crawling business logic and algorithms application. Steps of run processing:

1. Gets the Batch serialized object from stdin.
2. For each BatchItem from Batch.urls list (md5 string URL identifier) – gets URL record from MySQL db table:
    - If BatchItem.siteId=="" the table is `dc_urls`.`urls_0`.
    - If BatchItem.siteId!="" then table is "`dc_urls`.`urls_`" + BatchItem.siteId.

   If record not found then insert new URL record in to the correspondent table with values from the batchItem.urlObj field contains the URL object filled from the db located on the source host by URL_FETCH operation.

   Farther, the table that contains correspondent record will be named just "urls", but depends on site type (general or not) it must be treated as `dc_urls`.`urls_0` or as `dc_urls`.`urls_%SITE_ID_MD5%` where is the "%SITE_ID_MD5%" it is a macro name that need to be replaced with value detected in some way.

   2a. Reset `urls`.`ErrorMask` to accumulate only newly error bits for current url.

3. From `dc_sites`.`sites` read correspondent record. If BatchItem.siteId=="" siteId="0".
4. If SITE.State is not "Active" then skip this BatchItem and process next.
5. If SITE.Errors> SITE.MaxErrors or SITE.Resources>SITE.MaxResources then accumulate(add to existing bits) corresponding bit of SITE.ErrorMask, increment SITE.Errors, set TcDate=NOW(), set SITE.State="Suspended" and update all this fields of this site in correspondent site's table detected above. Then process next BatchItem.
6. Get site property "HTTP_HEADERS" from `dc_sites`.`sites_properties` for this site. If empty or no records – then use default headers from correspondent crawler-task_headers.txt file from config ini file.
7. Get site property "HTTP_COOKIE" from `dc_sites`.`sites_properties` for this site. If empty or no records – act the same way as above, file name crawler-task_cookie.txt.
8. Prepare HTTP request using URL.URL, headers and cookie.
9. Update urls.Status="Crawling", urls.Crawled++, urls.Batch_Id for correspondent url's record in MySQL db.
10. Wait URL.RequestDelay, msec.
11. Using prepared HTTP headers make HTTP request and get robots.txt content. If content found and not empty – parse them and store the set of rules in temporary container (list or map) for this batch URLs set. When the batch processing finished – this set of rules need to be removed. Check URL in robots.txt rules and if not succeeds – then set urls.State="Error", corresponding bit in ErrorMask fields for sites(add to existing bits) and urls(set only current error bits) tables, urls.Status="Crawled", and another related Site's and URL's fields like urls.UDate and process next BatchItem.
12. Using prepared request – make HTTP request and wait on response urls.HTTPTimeout msec.

13. If response got – detect: HTTP response code, Charset, MIME Content-Type, and so on downloaded resource's properties and HTTP response fields. If timeout reached – set the corresponding ErrorMask bit for urls.ErrorMask field, and other fields the same way as in case of robots.txt rule not matched case.
14. Update fields of this URL record urls.Status="Crawled", urls.State="Error" (if some error), urls.ContentType, urls.Charset, urls.ErrorMask, urls.TotalTime, urls.CrawlingTime, urls.UDate=NOW(),urls.HTTPCode to proper value.
15. If some other error like zero response, wrong response, HTTP error, content size greater thanMaxResourceSize and so on – set corresponding ErrorMask bit, update urls record.
16. If any kind error trapped that lead to zero content size – remove item from the Batch.items and process next item.
17. Check files directory and create if not found if BatchItem.siteId!="":
    - CONFIG_DATA_DIR/SITE_ID_MD5/PathMaker(RESOURCE_ID_MD5).getDir()/

    and if BatchItem.siteId=="":

    - CONFIG_DATA_DIR/0/PathMaker(RESOURCE_ID_MD5).getDir()/
18. Write HTML content file (if MIME Content-Type requires – converted to utf-8):
    - RESOURCE_ID_MD5_ASCIITIME.bin
19. Write HTTP response header file:
    - RESOURCE_ID_MD5_ASCIITIME.headers.txt
20. Write HTTP request file:
    - RESOURCE_ID_MD5_ASCIITIME.request.txt
21. Collect URLs from HTML "A", "IMG", "IFRAME", "FRAME" and so on, complete list see here "http://stackoverflow.com/questions/2725156/complete-list-of-html-tag-attributes-which-have-a-url-value/" and canonize them to have only fully qualified URLs uses "HTTP" protocol only (later another protocols support will be added).
22. For each collected URL check if already exists by URLMD5 field and if not – insert record to the urls table detected on step 2) (general or corresponds to this site), set fields with proper values of Site_Id, URL and so on. The Type, RequestDelay, HTTPTimeout set from correspondent Site fields. Fields that not defined leave default value. The URL candidate to insert need to be verified as belonged to this site by usage `dc_sites`.`site_filters` table records by `Site_Id`. The record has `Pattern` field defines the regexp pattern to check and the `Type` field defines check result treat as exclude this URL from farther processing (0) or include (1). The URL candidate to insert in to the database need to be checked by all filters for this site and in case of satisfy (no one pattern will exclude it) need to be inserted. If some check is not success – the URL skipped and not inserted in to the database for farther processing.
23. Serialize the resulted Batch object (possible, some items were removed) and print serialized string content to the stdout. Finish with zero exit status value in case of no critical errors and with 1 in another case. The error that can be treated as critical related with some DB structure, file I/O operations like permissions, wrong input object and so on…

Site.ErrorMask and URLs.ErrorMask specification

Bit #	Description
Crawling errors	

0	Wrong URL
1	Timeout
2	HTTP error
3	Empty content
4	Wrong MIME type
5	Connection error
6	Code page convert error
7	Bad redirection
8	Size error
9	Authorization error
10	File operation error, write file, create directory and so on
11	Robots.txt rule not matched.
12	HTML_PARSE_ERROR
13	BAD_ENCODING
14	SITE_MAX_ERRORS
15	SITE_MAX_RESOURCES
16	RAW_CONTENT_NOT_STORED
17	MAX_ALLOW_HTTP_REDIRECTS
18	MAX_ALLOW_HTML_REDIRECTS
Processing errors	
32	MaxErrors limit reached
33	MaxResources limit reached
34	Unsupported content-type for processing
35	Error raw content decoding
36	Scraper processing error
22	
23	

24	
25	
26	
27	
28	
29	
30	

Processor-task application

This application represents sets of data processing algorithms that can be used to process locally stored data. Located and acts on data node. This is main an after crawling data processor business logic and algorithms application. Steps of run processing:

1. Gets the Batch serialized object from stdin.
2. For each BatchItem from Batch.urls list (md5 string URL identifier) – gets URL record from table:
    - If BatchItem.siteId==”” the table is `dc_urls`.`urls_0`. Site Id=”0”.
    - If BatchItem.siteId!=”” then table is “`dc_urls`.`urls_`” + BatchItem.siteId.
3. From `dc_sites`.`sites` read correspondent record. If BatchItem.siteId==”” Site Id=”0”.
4. If SITE.State is not “Active” then skip this BatchItem and process next.
5. If SITE.Errors> SITE.MaxErrors or SITE.Resources>SITE.MaxResources then set corresponding bit of SITE.ErrorMask, increment SITE.Errors, set TcDate=NOW(), set SITE.State=”Suspended” and update all this fields of this site. Then process next BatchItem.
6. Form `dc_urls`.`urls_SITE_ID_MD5` table read corresponded record by the `URLMd5` field value.
7. From `dc_sites`.`sites_properties` read all records belongs this site by the `Site_Id` field. For all records with field `Name`=”PROCESS_CTYPES” check is value of field `dc_urls`.`urls_SITE_ID_MD5`.`ContentType` the same as one of `dc_sites`.`sites_properties`.`Value`. If not, skip this URL, set correspondent `ErrorMask` as “Unsupported content-type for processing”, update URL record and go to process next item from the batch.
8. Update URL.Status=”Processing”, URL.Processed++ for correspondent record.
9. Call processing algorithms methods and create resulted object.
10. Serialize resulted object to the json format.
11. If BatchItem.siteId!=”” – connect to the site file db using path:
    - CONFIG_DB_DIR/SITE_ID_MD5.db

    If BatchItem.siteId==”” – connect to the site file db using path:
    - CONFIG_DB_DIR/0.db
12. Insert or update the record, key is URL_MD5_ID, value is json string, CDate is NOW().
13. Update fields of this URL record URL.Status=”Processed”, URL.State=”Error” (if some error), URL.ErrorMask=URL.ErrorMask & “Error mask” (if some error), URL.TotalTime=URL.TotalTime+time, URL.ProcessingTime=time, URL.UDate=NOW() to proper value.
14. Update `dc_sites`.`sites`.`Contents` + 1 for current siteId.
15. Serialize the resulted Batch object and print serialized string content to the stdout. Finish with zero exit status value in case of no critical errors and with 1 in another case.

Db-task application

The application that acts as a main interface with DTM application to distributed DB units. The DB units manage the local databases of all objects and items defined on local data node level. Also, this application fetches locally stored resources. Located and acts on data node. The main SQL database schemas defined in the dc_sites.sql and dc_sites_template.sql dump files.

This is main data storages (SQL DB, key-value DB and files) operations business logic and algorithms application. Steps of run processing:

1. Gets the serialized object from stdin. This object it is tuple of (OPERATION_CODE, OperationObject).
2. Qualify the object type by one of two operations groups:
   - **Site: (SITE_NEW,Site), (SITE_UPDATE, SiteUpdate), (SITE_STATUS, SiteStatus), (SITE_DELETE, SiteDelete), (SITE_CLEANUP, SiteCleanup)**
   - **URL: (URL_NEW, list(URL)), (URL_STATUS, list(URLStatus)), (URL_UPDATE, list(URLUpdate)), (URL_DELETE, list(URLDelete)), (URL_FETCH, list(URLFetch)), (URL_CLEANUP, list(URLCleanup)), (URL_CONTENT, list(URLContent)), (URL_PURGE, list(URLPurge)), (URL_AGE, list(URLAge))**
3. Step depends on operation code and object type. Object types and operation codes:
   - **SITE_NEW::Site** – Insert new site operation. Steps:
     i. Check is site already exists by Id, if yes – return error and move to step ix If no – move to step ii.
     ii. Insert record in to the table `dc_sites`.`sites`.
     iii. Create table `dc_urls`.` urls_SITE_ID_MD5`.
     iv. For each item of Site.urls insert record in to the `dc_sites`.`sites_urls` table.
     v. For each item of Site.urls insert record in to the `dc_urls`.` urls_SITE_ID_MD5` table.
     vi. For each Site.filters item insert record in to the `dc_sites`.`filters` table.
     vii. For each Site.properties item insert record in to the `dc_sites`.`properties` table.
     viii. If ‘Content storage types: 0’ in hce-node-bundle/api/python/ini/db-task.ini then move to ix, if ‘Content storage types: 1’ then skip ix and x, move to xi
     ix. Create file of sqlite db by copy from template file:
        a) CONFIG_INI_DIR/db-task_keyvalue_template.db to CONFIG_DB_DIR/SITE_ID_MD5.db
     x. Create directory CONFIG_DATA_DIR/SITE_ID_MD5
     xi. Create table `dc_contents`.` contents_SITE_ID_MD5`.
     xii. Create GeneralResponse object and fill error code and states if some errors.
     xiii. Print serialized object to stdout.

   - **SITE_UPDATE::SiteUpdate** – updates fields of site table and correspondent records of related tables in the `dc_sites` database.
     i. Check is site already exists by Id, if yes – return error 2020 and move to step iv, If no – move to step ii
     ii. If site property UPDATE_NOT_INSERT_ROOT_URLS exists and !=0 then delete root URLs from dc_urls.urls_SITE_ID_MD5 and copy records from dc_sites.sites_urls to dc_urls.urls_SITE_ID_MD5. Otherwise skip this step

iii. If field of the SiteUpdate object is not None and it is single value – then update it with new value determinated in base json. If it is container (urls, filters, properties) than behavior depends on the SiteUpdate.updateType field.
    a) If it is UPDATE_TYPE_APPEND (updateType=0) than correspondent Site_Id records need to be added to tables `dc_sites`.`sites_filters`, and so on
    b) If it is UPDATE_TYPE_OVERWRITE (updateType=1) than correspondent Site_Id records need to be removed and insert new to tables `dc_sites`.`sites_filters` and so on
    c) If it is UPDATE_TYPE_UPDATE (updateType=2) then update existing fields with a new value
iv. If SiteUpdate.state=STATE_RESTART (dc_sites`.`sites'.State = 6) than dc_sites.sites_urls records need to be copied to the dc_urls.urls_SITE_ID_MD5 table with default fields values for not matched fields; `dc_sites`.`sites`.State value need to be set as STATE_ACTIVE, also, fields UDate=NOW, Iterations=Iterations+1, ErrorMask=0 and Errors=0.
v. Create GeneralResponse object and fill error code and statuses if some errors.
vi. Print serialized object to stdout.

- **SITE_STATUS::SiteStatus** – used to get registered Site object status. Returns the Site object filled with values from DB tables. Steps:
    i. Create new Site object.
    ii. Select correspondent record from the `dc_sites`.`sites` table by SiteStatus.siteId=`dc_sites`.`sites`.`Id` condition. If record not found, filled Site.State=STATE_NOT_FOUND value and go to last step.
    iii. Fill correspondent Site object single values fields with values from db record.
    iv. Fill correspondent Site object container fields (urls, filters, properties) with values from correspondent tables by <TABLE>.Site_Id=SiteStatus.siteId condition.
    v. Print serialized Site object to stdout.

- **SITE_FIND::SiteFind** – used to find registered site and return the Site object filled with values from DB tables
    i. Create new Site object
    ii. Select correspondent record from the `dc_sites`.`sites` table by SiteStatus.siteId=`dc_sites`.`sites`.`Id` condition. If record not found, return empty itemObject
    iii. Fill correspondent Site object single values fields with values from db record.
    iv. Fill correspondent Site object container fields (urls, filters, properties) with values from correspondent tables by <TABLE>
    v. Create GeneralResponse object and fill error code and statuses if some errors.
    vi. Print serialized object to stdout.

- **SITE_DELETE::SiteDelete** – used to delete registered site and remove all data from local storages. Steps:
    i. Delete correspondent record from the `dc_sites`.`sites` table by SiteStatus.siteId=`dc_sites`.`sites`.`Id` condition.

ii. Delete correspondent records from`dc_sites`.`sites_urls`, `dc_sites`.`sites_filters` and the `dc_sites`.`sites_properties` tables by Site_Id.
iii. Drop table `dc_urls`.` urls_SITE_ID_MD5`.
iv. Drop table `dc_urls_deleted`.urls_SITE_ID_MD5`
v. If "delayedType" in base json is not determinate – then it default value is “1”, move to vi; if "delayedType" value is “0” then skip steps vi-ix and move to x
vi. If ‘Content storage type: 0’ in hce-node-bundle/api/python/ini/db-task.ini then move to vi, if ‘Content storage type: 1’ then skip vi and vii; move to viii
vii. Delete file of sqlite db by name:
    a) CONFIG_DB_DIR/SITE_ID_MD5.db
viii. Delete directory by name:
    a) CONFIG_DATA_DIR/SITE_ID_MD5
ix. Drop table `dc_contents`.` contents_SITE_ID_MD5`
x. Create GeneralResponse object and fill error code and statuses if some errors.
xi. Print serialized object to stdout.

- **SITE_CLEANUP::SiteCleanup** – used to delete data of registered site from local storages. Steps:
    i. If "delayedType" in base json is not determinate – then it default value is “1”, move to ii; if "delayedType" value is “0” then skip ii, iii and move to iv
    ii. Create if not exist table `dc_urls_deleted`.urls_SITE_ID_MD5`
    iii. If parameter “saveRootUrls’ in base json = 0 then copy all records from `dc_urls.urls_SITE_ID_MD5 to `dc_urls_deleted`.urls_SITE_ID_MD5`; if saveRootUrls’=1 then copy only non-root records from `dc_urls.urls_SITE_ID_MD5 to `dc_urls_deleted`.urls_SITE_ID_MD5
    iv. Parameter “saveRootUrls’ in base json = 1 then delete records with not empty ParentMD5 from `dc_urls.urls_SITE_ID_MD5 (delete only non-root URLs) and skip step v; if it is 0 then move to step vi
    v. Truncate table `dc_urls`.` urls_SITE_ID_MD5`.
    vi. If "delayedType" in base json is not determinate – then it default value is “1”, move to vii; if "delayedType" value is “0” then skip vii, viii, ix and move to x
    vii. If ‘Content storage type: 0’ in hce-node-bundle/api/python/ini/db-task.ini then move to iv, if ‘Content storage type: 1’ then skip iv; move to v
    viii. Delete directory CONFIG_DATA_DIR/SITE_ID_MD5 and sqlite db by name CONFIG_INI_DIR/db-task_keyvalue_template.db
    ix. Drop table `dc_contents`.` contents_SITE_ID_MD5`
    x. Update correspondent `dc_sites`.`sites` record with fields: TcDate=NOW, Resources=0, Iterations=0, State=STATE_ACTIVE, ErrorMask=0, Errors=0.
    xi. If parameter “moveUrls” in base json = 1 then copy records form `dc.sites`_`sites.url` to `dc_urls`_urls_SITE_ID_MD5; if it is 0 then skip this step and move to viii
    xii. Create GeneralResponse object and fill error code and statuses if some errors.
    xiii. Print serialized object to stdout.

- **URL_NEW::list(URL)** – used to add list URLs to the system for sites or as not linked with sites, or with creation of new site if not found by Id or canonized domain name. Steps:
    i. Create GeneralResponse object.
    ii. For each URL object from list set fields that depends from the site if has None value (RequestDelay, HTTPTimeout, URLType):

a) If URL.siteId!="" – select record from the `dc_sites`.`sites` table;
   - If record not found then behavior process step iii and get fields value from correspondent record;
   - If record found – get fields values from this record and process step iv.

b) If URL.siteId=="" or URL.siteId==None then process step iii.

iii. Depend on the URL.siteSelect value:

a) If SITE_SELECT_TYPE_EXPLICIT:
   - If record not found or URL.siteId=="" then insert record in to the `dc_urls`.`urls_0` table.

b) If SITE_SELECT_TYPE_AUTO:
   - Qualify URL.URL (use the: scheme+netloc+"\" operation to generate canonic url form), generate SITE_ID_MD5 from qualified url and try to resolve site in `dc_sites`.`sites` table. If resolved, repeat step ii.
   - If site not resolved, create new site with fields values from dc.EventObjects.Site object and insert URL to the newly created table in `dc_urls`.`urls_SITE_ID_MD5` table. As single root URL use the scheme+netloc+"\" canonization and initialize new Site object with this url.

c) If SITE_SELECT_TYPE_QUALIFY_URL:
   - Qualify URL.URL, generate SITE_ID_MD5 and try to resolve site in `dc_sites`.`sites` table. If not resolved, put URL to general DB table `dc_urls`.`urls_0`.

iv. Check is URL exists for this site and if not – insert URL in to the correspondent urls table. If URL already exists set GeneralResponse.states item value as 2. If some db error – set it as 1.

v. Insert item in to the GeneralResponse.states with correspondent value.

vi. Print serialized results list object to stdout.

- **URL_STATUS::list(URLStatus)** – used to get status of list of URLs. Steps:

i. Create results list.

ii. For each URLStatus object in the list: create new URL object, select record from table `dc_sites`.`urls_URLStatus.siteId`, fill correspondent fields, put Site object in to the results list. The record selected by URLStatus.url field. If the URLStatus.urlType==URL_TYPE_MD5 the `dc_sites`.`urls_URLStatus.siteId`.`URLMd5`=URLStatus.url condition used, else – the `dc_sites`.`urls_URLStatus.siteId`.`URL`=URLStatus.url condition used.

iii. Insert URL object in to the list filled with correspondent field values.

iv. Print serialized results list object to stdout.

- **URL_UPDATE::list(URLUpdate)** – used to update list of URLs with new values. Only fields with not None values are updated. Steps:

i. Create GeneralResponse list.

ii. For each URLUpdate objects in list using URLUpdate.url==<TABLE>.URLMD5 condition:

a) If URL.siteId=="" – update record in the `dc_urls`.`urls_0` table;

b) If URL.siteId=="" – update record in the `dc_urls`.`urls_SITE_ID_MD5` table.

iii. Insert the GeneralResponse.statuses item with correspondent value.
iv. Print serialized object to stdout.

- **URL_DELETE::list(URLDelete)** – used to delete record correspondent to URL and data in file system and key-value DB. Steps:
  i. Create GeneralResponse list.
  ii. For each URLDelete objects in list:
    a) If URLDelete.siteId!=""
      - delete record in the `dc_urls`.`urls_SITE_ID_MD5` by URLDelete.siteId with `dc_urls`.`urls_SITE_ID_MD5`.`URLMd5`=URLDelete.url condition;
      - Delete item in the key-value db file: CONFIG_DB_DIR/"URLDelete.siteId".db by URLDelete.url key;
      - Delete files in directory by mask: CONFIG_DATA_DIR/URLDelete.siteId/PathMaker(URLDelete.url).getDir()/URLDelete.url*

    b) If URLDelete.siteId==""
      - delete record in the `dc_urls`.`urls_0` by URLDelete.siteId with `dc_urls`.`urls_0`.`URLMd5`=URLDelete.url condition;
      - delete item in the key-value db file: CONFIG_DB_DIR/0.db by URLDelete.url key;
      - delete files in directory by mask: CONFIG_DATA_DIR/0/PathMaker(URLDelete.url).getDir()/URLDelete.url*

  iii. Decrement `dc_sites`.`sites`.Resources and `dc_sites`.`sites`.Contents in case of each record are deleted from `dc_urls`.`urls_SITE_ID_MD5` and from key-value db.
  iv. Insert the GeneralResponse.statuses item with correspondent value.
  v. Print serialized object to stdout.
- **URL_CLEANUP::list(URLCleanup)** – used to update record correspondent to URL with value State=0, Status=1 and delete raw data in file system and key-value DB. Steps:
  i. Create GeneralResponse list.
  ii. For each URLCleanup objects in list:
    a) If URLCleanup.siteId!=""
      - update fields "state" and "status" of record record in the `dc_urls`.`urls_SITE_ID_MD5` by URLCleanup.siteId with `dc_urls`.`urls_SITE_ID_MD5`.`URLMd5`=URLCleanup.url condition;
      - delete item in the key-value db file: CONFIG_DB_DIR/"URLDelete.siteId".db

      by URLDelete.url key; delete files in directory by mask:

      - CONFIG_DATA_DIR/URLDelete.siteId/PathMaker(URLDelete.url).getDir()/URLDelete.url*

b) If URLDelete.siteId=="" – update fields "state" and "status" of record in the `dc_urls`.`urls_0` table by URLDelete.url=`dc_sites`.`sites`.`URLMD5` condition; delete item in the key-value db file:
   - CONFIG_DB_DIR/0.db

   by URLDelete.url key; delete files in directory by mask:
   - CONFIG_DATA_DIR/0/PathMaker(URLDelete.url).getDir()/URLDelete.url*

iii. Insert the GeneralResponse.statuses item with correspondent value.
iv. Print serialized object to stdout.

- **URL_FETCH::list(URLFetch)** – used to fetch urls from correspondent tables. Steps:
  i. Create results list.
  ii. If URLFetch.sitesList is empty or None, select all records from the `dc_sites`.`sites` table with condition state="Active" and fill URLFetch.sitesList with Id value.
  iii. For each item in the URLFetch.sitesList:
    a) Select records from table `dc_urls`.`urls_%URLFetch.siteId[i]% with conditions using URLFetch.criterions items.
    b) Create new URL object and fill fields' values from selected record.
    c) Insert URL object in to the results list
  iv. Print serialized object to stdout.
- **URL_CONTENT::list(URLContentRequest)** – used to fetch resources corresponded to URL specified. Steps:
  i. Create results list object.
  ii. For each URLContentRequest object in list:
    a) Create new URLContentResponse object.
    b) If URLContentRequest.contentTypeMask has CONTENT_TYPE_PROCESSED bit set – return the processed content from correspondent key-value db:
      - If URLContentRequest.siteId!="" than use db file: CONFIG_DB_DIR/"URLContentRequest.siteId".db
      - If URLContentRequest.siteId=="" than use db file: CONFIG_DB_DIR/0.db

      and the md5(URLContentRequest.url) as key. Create new Content object, fill fields values and insert it in to the URLContentResponse.processedContents as item.
    c) If URLContentRequest.contentTypeMask has CONTENT_TYPE_RAW_LAST, CONTENT_TYPE_RAW_FIRST, CONTENT_TYPE_RAW_ALL bit(s) set – return correspondent raw content(s) from file by mask:
      - If URLContentRequest.siteId!="" than use directory: CONFIG_DATA_DIR/URLContentRequest.siteId /PathMaker(URLContentRequest.urlMd5).getDir()/
      - If URLContentRequest.siteId=="" than use directory: CONFIG_DATA_DIR/0/PathMaker(URLContentRequest.urlMd5)) .getDir()/

Read content of correspondent raw data file, create new Content object, fill fields values and insert it in to the URLContentResponse.rawContents as item.

iii. Insert URLContentResponse object in to the results list.
iv. Print serialized results list object to stdout.

4. Finish with zero exit status value in case of no critical errors and with 1 in another case.

Sites manager

The main client interface for Site and URL objects operations

Application configuration and start

Threaded classes instantiation sequence

The threaded classes use inproc server and client connections. Inproc connections need to have server-side ready (bind and listen) to make connect operation from client side. This condition requires some instantiation sequence for threaded classes because they make connections at this time. The instantiation sequence:

1. AdminInterfaceServer.
2. URLManager.
3. SitesManager.
4. BatchTasksManager.
5. ClientInterfaceService.

Command line arguments

DCDaemon dc-daemon.py

- --config, -c – configuration file path name. Mandatory parameter, if application cant to load config file it exit(1) with corresponding error message on console, no demonization, no application functionality started. If omitted application tries to open file with the same name as executable module and ".ini" extension in sequence:
    - in the current directory,
    - in the /etc/<application_binary_name>/ directory,
- --help, -h – displays brief usage information, including possible command line arguments names and values and exit(0).
- --name, -n – application instance name. Optional parameter, if omitted the same name as executable module file used. Overrides the same parameter in the configuration file "Application" section.

DCClient dc-client.py

- --config, -c – the same as for the "dtmd" application.
- --help, -h – the same as for the "dtmd" application.
- --command, -cmd – task action, specifies the requested task operation, possible values {"SITE_NEW", "SITE_UPDATE", "SITE_STATUS", "SITE_DELETE", "SITE_CLEANUP", "URL_NEW, "URL_STATUS", "URL_UPDATE", "URL_FETCH", "URL_DELETE", "URL_CLEANUP", "URL_CONTENT"}. Optional if "--help" is specified, and mandatory – if not. A result of any action is file in json format. Structure corresponds with DRCE Functional Object response specification.
- --file, -f – data file for action, specified by "--task" option. Mandatory if "--task" is specified. The data file in json format, structure defined by DRCE functional object request protocol specification.
- --fields, ff – fields set json string that will be used after target object created from pre-created json file (--file parameter) or from regular construction as empty filled with None values all fields. More detailed see the DC_public_client_API document.

- --merge – specifies merge or not results received from different hosts in multi-host distributed cluster. By default if not specified by user the Boolean true or greater than zero value that is means – merge. Boolean false or zero value – means not merge and return the array of results from each node.

DCAdmin dc-admin.py

- --config, -c – the same as for the “dtmd” application.
- --help, -h – the same as for the “dtmd” application.
- --cmd – command name, specifies the requested admin management operation. Possible values are {“STAT”, “SET”, “GET”, “STOP”}. “STOP” – lead to sequentially stop all threaded classes of application in opposite start sequence (specified in the “Server” section by the “instantiateSequence” option of configuration file) and application exit(0).
- --fields – comma separated fields list for requested operation. Mandatory, if “--cmd” is set and is not a “STOP” or “STAT” value. For “STAT” – specifies list of statistical indicators for class handler. If empty or not set – the empty list of fields used. For “GET” – specifies names of configuration options for all threaded classes and application that can be changed at runtime. For “SET” – specifies list of configuration options name:value pairs for all threaded classes and application that can be changed at runtime. All commands returned json file content. The structure depends on concrete request.
- --classes – comma separated list of threaded classes that will be used to execute request. Always optional for “STOP” command (always executed for all) and mandatory for all another commands. For “STOP” if omitted or empty – the list specified by the “instantiateSequence” option in “Server” section of configuration file.