The Local Content Cache

1. Overview

The goal of the Local Content Cache (LCC) pilot project is to create a functioning proof-of-concept of a system that can be used to cache url and related content.

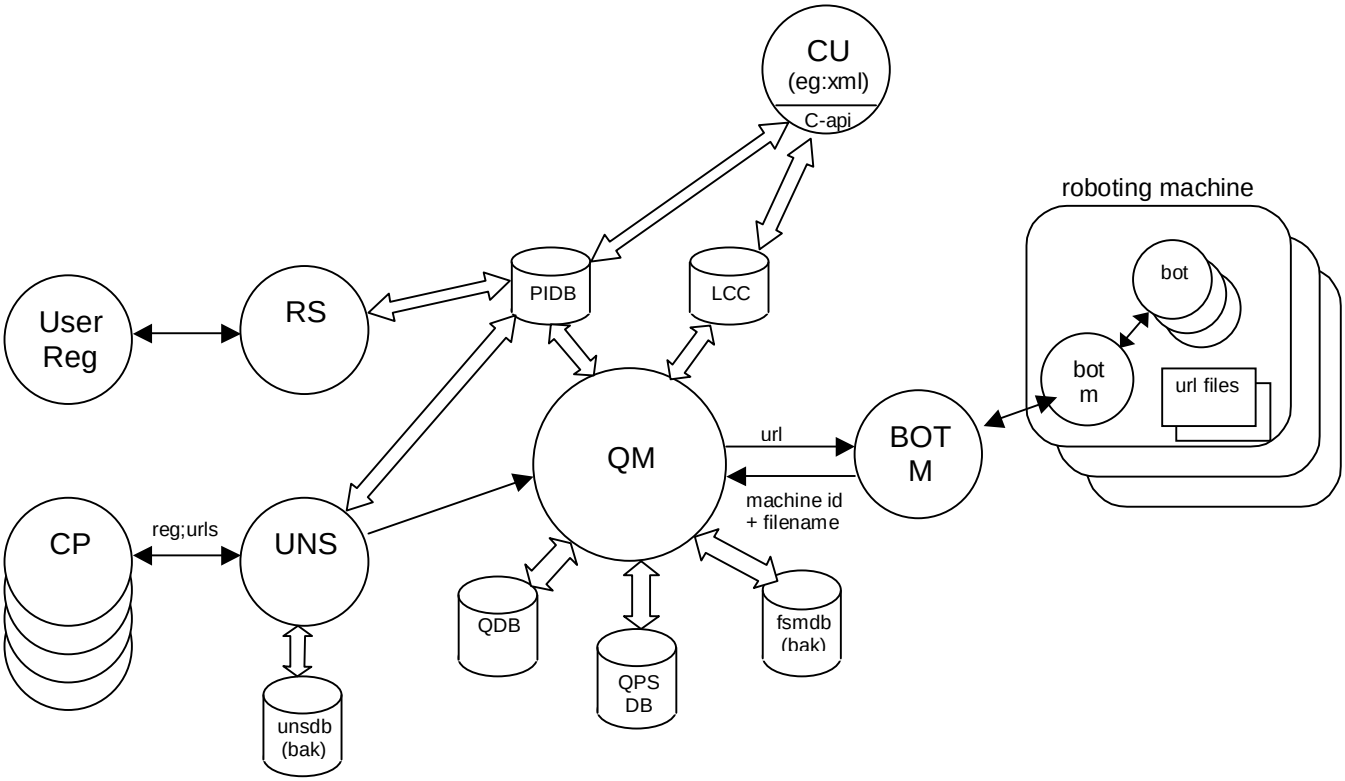
Instead of, as is found with traditional search engine systems, continually polling all contributing sites for data, the LCC is oriented to an “update notification” approach. In this approach, individual “content providers” each connect to a central site when they themselves have determined that local content has actually changed. These content providers transmit the url information corresponding to the changed information to the Local Content Cache which, at some later time, fetches and stores the content.

In addition to simply storing direct copies of urls, it is possible to store various “views” of the data. In this manner a given content provider can submit raw html, XML and search-engine-processed versions of the same underlying url, for example. The work of performing these transformations is distributed amongst the content providers rather than having to be performed only at the central LCC site.

Sometime later, of course, this stored data can be used. A “content user” can interrogate the LCC using various parameters to retrieve sets of data (urls and optionally content); eg, finding everything that has been reported modified since a certain date. The initial proof-of-concept LCC will demonstrate use by some specific content users such as ht://Dig, SWISH++ and the NexTrieve search engine.

2. General Technical Overview

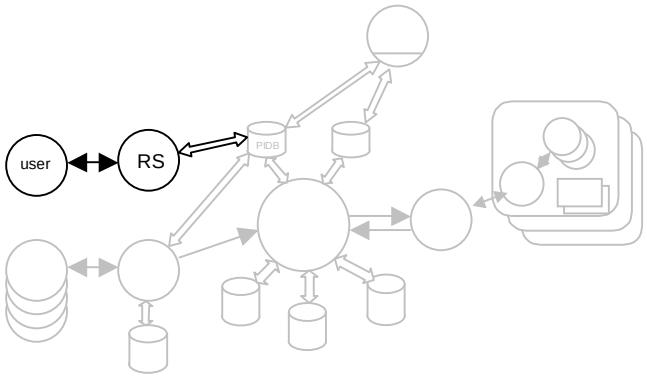
The following diagram shows the main processes in the LCC.



Following sections discuss each of the above processes and protocols in detail, but the main processes can be summarised as follows:

User Reg: A person performing the initial registration process.
RS: The Registration Server.
CP: A Content Provider submitting URL information.
UNS: An Update Notification Server. This processes accepts sets of URLs from Content Providers.
UNSDB: Update Notification Server Database. This is used if the QM is not available.
PIDB: The Provider Information Database.
QM: The Queue Manager. This process orchestrates URL fetching.
QDB: The Queue Database. There is one record per URL to be fetched.
QPSDB: The Queue Provider Set Database. This defines an order for URL fetching.
FSMDB: Free Space Manager Database. This database is used if various BOT-M processes are not available.
LCC: The Local Content Cache Database. There is one record per URL in the system.
BOT-M: A Robot Multiplexer.
BOT: A URL fetching Robot.
CU: A Content User.

3. Initial User Registration



The Registration Server allocates the provider number and eventually initializes a new row in the PIDB (Provider Information Database) recording this and other provider information. A simple authentication procedure is used.

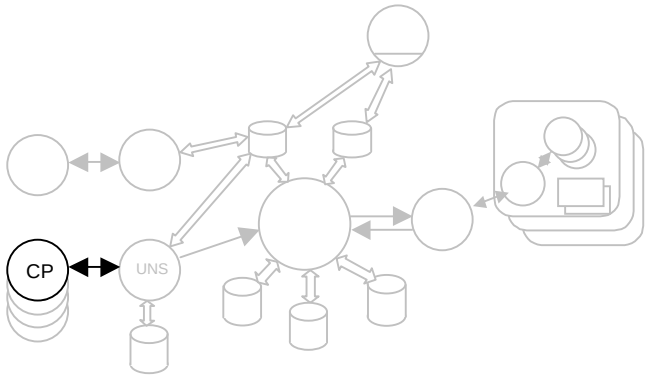
The authentication procedure merely establishes that the content provider is in a position to alter the website for which it manages, nothing more. The RS determines this by providing a “secret” to the user that must be placed in a file to be retrieved on the provider’s website in as high a position in the website tree as possible. After this “secret” has been retrieved by the RS the content provider may then provide url notifications for trees that match the same directory prefix (from a URL perspective) of the file that contained the secret.

A secret and a provider password are allocated in pairs, retrieval of the secret effectively activates the password and thus allows the content provider to start sending notifications. The secret/password pair are provided by means of a simple cgi script and webpage and have an expiry time associated with them, after which the details are removed from the system if not activated.

The registration process is thus a two-step process. Step one allocates the secret/password pairs, optionally sending this information in an E-mail if requested, in addition to displaying on the screen. Step two is where the content provider registrer comes back to the website and signifies that the secret has been placed the specified location. The RS then attempts to retrieve this file immediately and reports success or failure. Once verified the registered the file containing the secret is no longer referenced or needed and can be safely deleted if necessary.

Only after the user receives a success indication from the registration website may the content provider notification processes be activated.

4. CP: The Content Provider



A few notes about the Content Provider:

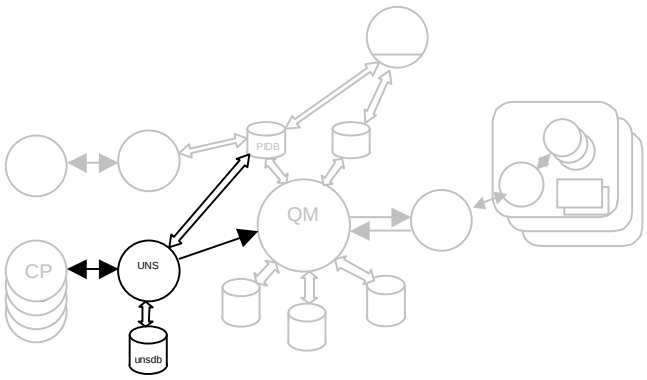
- The CP only works if registration has been performed.
- The CP connects to a UNS:
    i. periodically to check that the LCC has not expired any files over a period of months.
    ii. whenever something has changed on the local system that the sysadmin wants reflected in the LCC (this might be done periodically as well, rather than instantly; up to the sysadmin).

When the CP wants to send some modification notifications, it performs the following operations:

- Before connecting, the CP generates the modifications that it wants to send.
- After connecting, the CP sends its registration information.
- The CP then receives from the LCC:
    i. A "send full set" flag, plus a reason if it is set (cache broken, some files are > 3 months old, some files were > 6 months old and have been deleted).
    ii. A set of mime types that are accepted by the LCC.
    iii. A quota limit, and how much space is currently used in the LCC by this CP (our CP will perform local processing to try to ensure the LCC quota is not overflowed).
    iv. The number of unsolicited "full sets" that can still be sent.
    v. One or more error containers (up to a maximum number, configurable in the LCC) of urls from previous CP connections that have encountered processing errors after being sent to the LCC, along with the error reason. A complete error count (which may be greater than the number of error containers) is also returned.

- If the CP receives a "send full set" indication, it is at liberty to “go away” and come back later with a full set if generating a full set is an expensive or time-consuming operation. It is also at liberty to still send its update notifications, and send the full set later.
- The CP sends a "this is a full set" indication (on or off).
- The CP then sends a sequence of url elements, each containing:
    i. Conceptual URL (a “content id”).
    ii. Fetch URL (blank for “deleted”, perhaps).
    iii. Browse URL.
    iv. Mime type and subtype information (a subtype is a small textual field, verified only to be of the form "type:stuff", otherwise it is blanked).
    v. Time-of-day/week to fetch.
    vi. Other related information, such as modification time, file size and md5-checksum.
- It should be noted that a given url in the LCC is uniquely identified by the CP that submitted it, the “conceptual url” and the mime-type of the url.
- Each url information container can cause the UNS to return an error container giving the reason for the error, and the url information is ignored for that container. If the url is “OK” at this point, nothing is returned by the UNS to the CP.
- The CP then sends an end tag indicating the end of the session.
- The UNS will then send and ending container back to the CP, with an OK or NOT-OK response. If NOTOK, it implies that everything that the CP has sent has been ignored, otherwise everything has been initially accepted for "later processing".

5. UNS: The Update Notification Server



Overall Mission

The UNS module is the "front end" of the LCC. A CP wishing to update, add or delete content in the LCC connects to the UNS for that LCC to submit the information.

The UNS verifies that the CP is registered and then provides some system status information and any errors that have not yet been reported to the CP. The UNS then expects to be sent a sequence of records indicating update/delete requests from that CP which it collects and forwards to the QM for processing as a single "provider set".

If the QM is not present, the UNS will, for a time, store incoming records in a temporary file (the "unsdb") for later forwarding to the QM when it does become available.

Some More Details

The UNS will be a multi-threaded process with the following threads:

- One thread per active CP connection.
- One thread reading the unsdb and transmitting its content to the QM. This thread is only active when the UNS notices that the QM has come back after being unavailable; in the meantime incoming url information has been stored in the unsdb.

Each UNS connection thread expects to read an XML stream (see the section "CP/UNS Protocol" below for more details) from the CP, and responds with another XML stream.

Initially, a CP must provide registration information that is verified by the UNS by looking up the details in the PIDB (Provider Information database). After this information is verified, the UNS responds with some LCC status information including:

- The set of valid mime types that are accepted by this LCC.
- A count of the total number of errors that were encountered during the processing of the last update set sent by this CP.
- A set of urls that were in error (at least one, but otherwise an LCC configurable number) for some of these errors, along with their error codes.
- A count of the number of urls still being processed from this CP.
- An indication that the LCC is expecting a "full set" of url information from this CP and the reason why (reasons include "old files", "expired files", "cache crash").
- A counter giving the number of "complete sets" that can still be returned by this CP if it so wishes before it is penalised.

This information allows the CP to decide what notifications to send. The UNS expects a small header (indicating whether the set of urls is a "complete set" representing everything on the provider, or an "update set", representing modifications only), followed by a series of records, one record per url update/delete notification.

Each record contains basically:

- A "conceptual url" or "content id". This is required.
- The mime type of the content. This is required.
- A "browse url" suitable for a browser link. If absent the “conceptual url” will be used. There will be some way to specify that the information is not browsable (ie, that there is no valid “browse url”).

- A "fetch url" from where the information can be fetched. If absent the "conceptual url" will be used if the information is to be fetched.
- An update/remove specification.
- An optional time of day specification as to when to download or when downloads are prohibited.
- A modification timestamp, file size and md5-checksum.

There is other information, but it is not important here.

The only verification the UNS performs at this stage is a verification that the mime type of each url is in the set permitted for loading into the LCC. For every record encountered that is in error, an error container is sent to the CP, but processing continues.

Valid records are collected up and forwarded to an appropriate QM by using a permanently open TCP connection to that QM. In the case where the QM is not present, the set of urls will be written to a backup unsdb for later forwarding to the QM when it becomes active.

If the unsdb has become too large, the UNS will send a "rejected" result to the CP with an appropriate reason (such as "try again later, system temporarily off-line").

If everything works as expected, the UNS will send an "accepted" result, and the CP can terminate.

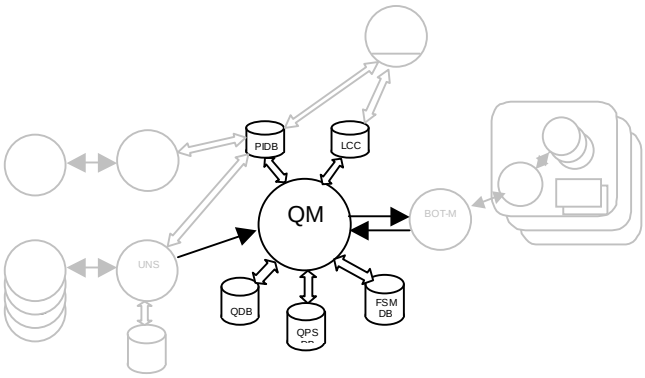
UNS Graceful Shutdown

If the QM is not present or is disallowing UNS connections, the UNSes can still function.

In such a case, incoming CP url sets are stored into a file, the "unsdb" in the general architecture diagram. Each UNS periodically checks to see if the QM is present and accepting connections. When the QM returns to an active state, each UNS will send url sets from its unsdb to the QM. During this time the UNS will also be appending url sets sent by CPs to this file until the file has been completely sent to the QM. After that point the UNS enters its normal "pass through" mode of operation, ie, incoming url sets from CPs are forwarded directly the QM.

If the UNS notices that its unsdb has grown beyond a certain configured limit, it will return an error of the form "system temporarily off-line, try again later" to any attempted CP connection.

6. QM: The Queue Manager



Overall Mission

The role of the QM is to coordinate the actual fetching of urls that have been received from UNSes. As such, this is the process that causes the bandwith/time-of-day constraints and provider priorities in url fetching to be obeyed, and also enforces url fetching from a particular provider to be performed in a *serial* fashion, rather than having multiple fetches in parallel.

The QM also updates the core LCC DB that contains information about the URLs loaded into the LCC; it is the only module that updates this database.

The QM also tries to ensure that a url is fetched only once, even if it has been the target of several update notifications (this may happen if the LCC is heavily loaded, for example, and a CP is sending update notifications of regularly changing pages).

The QM mainly interacts with:

- The LCC DB, to record the final state of fetch/remove operations.
- A QDB (Queue Database) that contains one record per url to be fetched.
- A QPSDB (Queue Provider Set Database) that has information regarding the ordering of the fetching of different provider set submissions.
- A BOT-M (Robot Multiplexer) process, to perform the fetching.

Some More Details

The databases used by the QM -- the QDB and QPSDB -- contain state that is preserved over QM restarts.

QDB Content

The QDB contains one record for every url that has arrived from a UNS. The record remains in the QDB until the update or delete operation for that url has been performed.

Each row in the QDB contains the following information:

URLKEY: A unique key built from the provider-id and conceptual-url fields of the URL to be fetched. This is used to remove duplicate submissions.
PID: The provider-id; might not be necessary.
PSETID: The provider-set-id; this is only used for reconstructing the QPSDB if required.
NEXTURLINSET: A URLKEY value representing the next URL to fetch; a blank value indicates this is the last URL in the set. Note that this field is not keyed in order to allow fast list update.
OPERATION: Either "fetch" or "nothing" ("nothing" for when a delete operation has come in for the url after a fetch for the url is already in the system.)
other: All the other information submitted by the UNS for this URL is also present in this record.

QPSDB Content

The QPSDB contains the provider-set ordering information. Every time a provider submits a bunch of urls, a monotonically increasing "provider set number" is incremented, giving the provider set id used here. The fetching of URLs is determined by the order of set submissions and, within each set, the order of URLs within the set.

There is normally a single record per provider set, although multiple records can occur where, for example, a single provider set has some urls that can only be fetched at certain times of the day and others that can be fetched at any time.

Each QPSDB row contains:

PSETID: The provider-set identifier, as a duplicate key. This represents the submission order of the set. It is a "duplicate" key because a single set might be split up during some operations.
PID: The provider-id.
STARTTIMESLOT: A time-of-day-delay slot number (if a url can only be fetched after a certain time), or a special value such as -1 representing "fetch now". This is a keyed field allowing duplicates.
PRI: Copy of the provider priority.
FIRSTURL: The QDB URLKEY of the first URL to fetch from this set.
LASTURL: The QDB URLKEY of the last URL to fetch from this set.

There can be at most one record in the QPSDB with any particular PSETID and STARTTIMESLOT combination.

Note that a “time slot” is something like “hour of the day” or “hour of the week” (the period “hour” and “day” or “week” being configurable in the LCC). This represents a rolling period of time. If the LCC is configured to have 24 slots, representing hours of the day, a CP can specify that urls are fetched between 0h and 8h on any day, for example, but cannot specify “only on sunday”. If the LCC is overloaded, the submitted urls may be fetched several days after submission, but only within the time-of-day specified.

It should be noted that the QPSDB can, in fact, be reconstructed from the information in the QDB. If the QM suspects some corruption of the QPSDB or suspects that the QDB and QPSDB are not in sync with each other, it is possible that it can rebuild the QPSDB from first principles by reading each QDB record.

QM Execution

The QM is multi-threaded process, with threads arranged in the following way:

- One thread per active UNS that submits url sets over a permanently open TCP connection. The arriving URLs are absorbed into a queue structure, described later.
- One or more threads dedicated to performing updates of the LCC DB and other databases.
- A thread dedicated to performing the fetch/remove processing of urls it takes from the head of a queue, interacting with a BOT-M process over a TCP connection.
- A thread dedicated to ensuring free-space for removed urls is actually removed, by interacting with the BOT-M process.
- A thread dedicated to merging the content of the next "time-of-day" delay queue with the "fetch-now" queue.

Arriving URLs

Every unique provider-id+conceptual-url keyed record that arrives is placed into the QDB. The provider-id+conceptual-url key is used to eliminate duplicate submissions and can be thought of as a "url key".

If a row already exists for the url key, this operation effectively performs duplicate removal by allowing the discovery that the same url has been submitted more than once. Only the last operation is remembered (recorded in the QDB) but is performed at the time the first operation would have been performed. These operations are normally expected to be update operations.

In the case where no row exists for the url key in the QDB, the QM looks in the LCC to see if the url is already present in the LCC. If it is, the timestamp, file size and md5-checksum are compared with the incoming url information. If none are different, the url is not fetched.

To absorb a url-set into the queue structure, the QM generates the next monotonically increasing provider-set-id value, and it can then perform the following two fundamental operations:

Insert an update notification:

The provider-id+conceptual-url is looked up in the QDB.
If the row exists, its status is set to "fetch", otherwise it is created with a status of "fetch".

Insert a removal notification:

The provider-id+conceptual-url is looked up in the QDB. If the row exists, its status is set to "nothing".
If the row does not exist, or had a status of "fetch", the LCC DB is examined to see if the url existed; if so, the entry is deleted, with information being passed to the delete-manager thread for file deletion.

If a QDB row was created in the "insert an update notification" case, the url must be linked in to a suitable record in the QPSDB for ordering purposes. If a record keyed by provider-set-id and appropriate fetch starting time-slot (or a special value for no particular fetch starting time) already exists in the QPSDB, a linked list of records already exists for this set. In that case, the QDB record just added is added to the end of this list by appropriate modification of the "lasturl" field of the QPSDB record, and the "nexturl" field of the QDB record that used to be the last.

If no QPSDB record is found, one is created with the QDB record just added being the only record in its list.

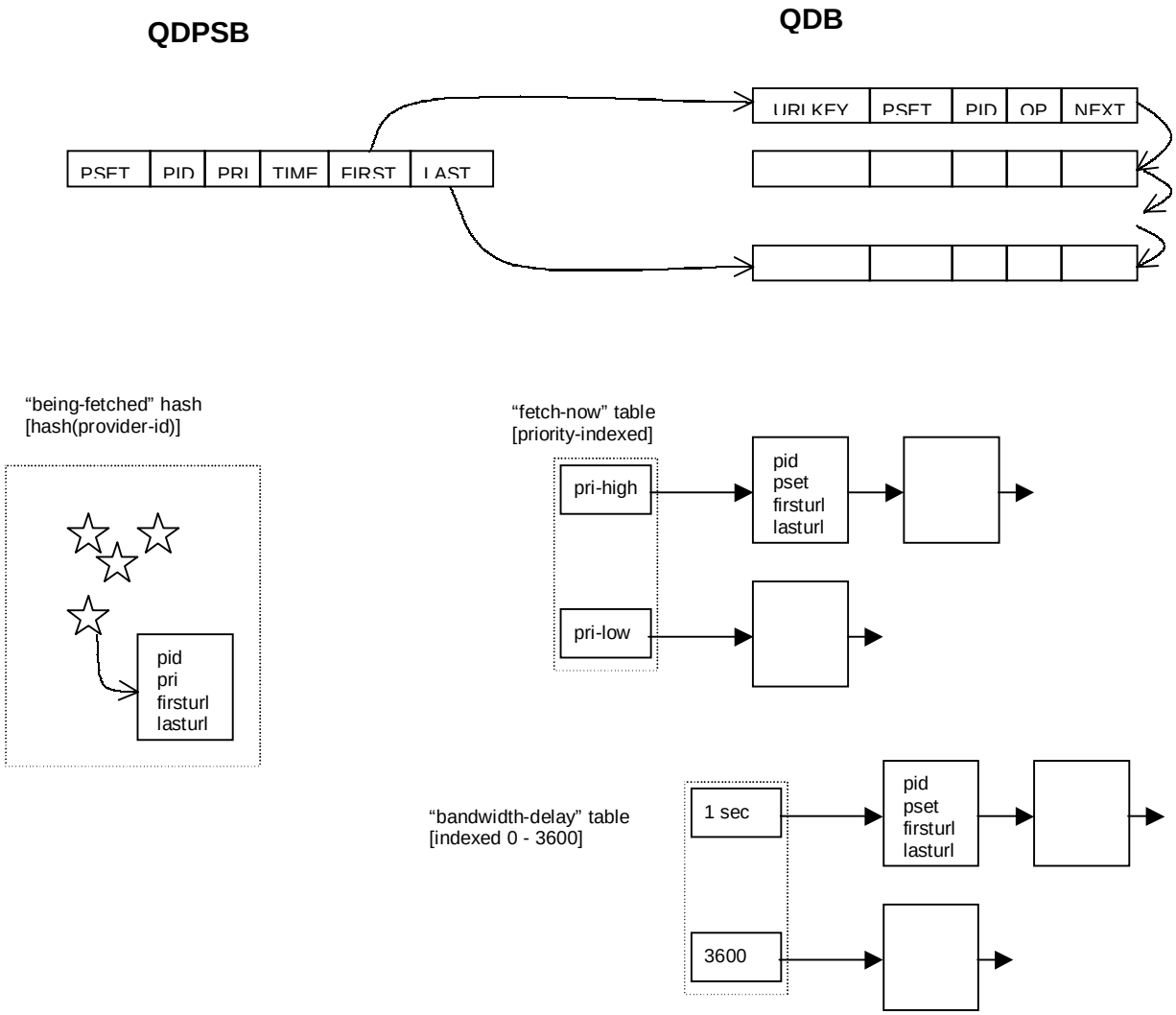
These operations are performed url-by-url when an update notification is sent from a provider.

In the case of a "full set" notification slightly more work has to be done, namely the removal of all urls in the LCC DB for this provider that are not in the passed set. This is simply done the QM going through the LCC DB for all records from this provider and, if the url is not in the passed set an "insert a removal notification" operation is performed to delete its content. This is followed by the normal update notification processing of the given set of urls.

Note that there will have to be an appropriate low-overhead synchronisation lock on these QM DB operations as they are performed here when data arrives, and are also performed by a fetch thread that is taking elements off the head of the QM queue and processing them.

Fetching of URLs

In QM memory, there is a record per provider being fetched from. This record can be present in one of the following structures:



- A "being-fetched" hash table, keyed by provider id. These records represent providers actually being fetched from.

- A "fetch-now" table with an entry per priority, each entry having a queue of provider structures that would like to have
- fetches performed when other active providers have finished.
- A "bandwith-delay" table where provider structures are moved temporarily when their bandwidth limits have been exceeded.

Initially, to get the set of providers from which to fetch urls, the following select can be done on the QPSDB:

SELECT * FROM QPSDB WHERE STARTTIMESLOT = -1 ORDER BY PSETID;

This gives the ordered sequence of sets from which to get urls to fetch. These records are absorbed into the "fetch-now" table (indexed by priority). There are of the order of #-providers of these records, normally a lot less, so this structure is intended to reside in memory.

To actively start to fetch things from providers, some provider-set records must be moved into the "being-fetched" hash table, hashed by provider id. The hash-table is filled from the "fetch-now" table, taking records from the highest priority queue first.

When a provider record is in the "being-fetched" hash table, it is an indication that it should be used to access URLs which are sent to the BOT-M process. When the BOT-M process eventually returns a response, the next URL to fetch from this provider will be accessed via this record.

To extract the next URL from a provider record, we can use the following select on the QDB:

SELECT * FROM QDB WHERE URL = firsturl
(getting urlkey, pid, psetid, nexturlinset)

Remember that the provider-set record is a copy of the record found in the QPSDB.

When the BOT-M returns a response it returns a url key, a machine-id and filename for a successful fetch and other interesting information (such as file size, timestamp, and md5 checksum). The QM then:

- Updates the LCC (updating or creating a record as necessary) with this information.
- Updates the local provider record so that its "firsturl" is the next URL to fetch.
- Updates the corresponding record in the QPSDB with something like:
  UPDATE QPSDB SET FIRSTURL=nexturl WHERE PSETID=psetid AND STARTTIMESLOT=-1;
- Updates the QDB to delete the URL record.

We also look at a bit of administration info (not discussed here) to see if this provider record in the "being-fetched" hash table should be moved back to the "fetch-now" table because a higher-priority provider has since come in. Otherwise, we submit the next URL from the set to the BOT-M.

If the fetch was in error, a reason code is returned. For some reasons it may be deemed suitable to retry all subsequent fetches from this provider at a later date (eg, a DNS type error), in which case the provider set is moved into an appropriate bandwith-delay or time-of-day-delay slot. If the error is specific to a url and is deemed to be retryable, the url record is moved to the end of the current provider set and marked "bad". If a "bad" record makes it to the front of the queue of the provider set it can either be retried, or the provider set can be moved to an appropriate bandwidth-delay or time-of-day-delay slot (depending on configured timings). If the error is deemed to be real (eg a real error, or number of retries exceeded), the QM will update the PIDB with the url and reason (if there is room in the provider entry),

and update the provider error counter for later reporting to a CP. The equivalent of an "insert removal notification" operation is performed on the url to delete its content from the LCC.

Time Of Day Fetching

Every hour on the hour (a configurable time), the QM can select other provider sets from the QPSDB that are now valid to fetch. It can do this with a select similar to:

SELECT * FROM QPSDB WHERE STARTTIMESLOT = st ORDER BY PSETID;

where "st" is the time-of-day-delay slot number corresponding to the current time of day.

This returns an ordered sequence of records which can be merged with the content of our "being-fetched" and "fetch-now" tables. We can do this by scanning the sequence returned by the select and:

- If the provider-id is in our "being-fetched" hash table *and* that record has the same provider-set-id as returned by the select, we will join the two URL lists up. We do this by changing the nexturl field of the last URL in the currently loaded set, and removing the newly discovered QPSDB record.
- Otherwise, we want to record the provider set queue in an appropriate "fetch-now" table entry, ordered by provider-set-id value. If we find a record already present with the same psetid value, we join the queues up as above, otherwise we simply set the record in the QPSDB to "active" with something like:
  UPDATE QPSDB SET STARTTIMESLOT = -1 WHERE STARTTIMESLOT = st AND PSETID = psetid;

When, during submission of a URL to the BOT-M, the QM notices the legal fetch time period has been passed for that URL, all invalid URLs are written back to another version of the set (ie, a new QPSDB record with an appropriate STARTTIMESLOT value). Further details are not gone into here.

Bandwidth Delay

When an "OK" response is received from a BOT-M and there are more URLs to submit from that provider, the QM will perform a bandwidth calculation. If it notices that the bandwidth has exceeded a provider-defined limit, the provider record will be moved into the appropriate slot in a "bandwidth-delay" table. This table can hold one-hour's worth of delay at 1-second intervals (both these periods being configurable), eg, 3600 entries. If the delay is too great (ie, over an hour), the provider set being fetched will be placed into the appropriate time-of-day bucket to be fetched later.

QM Delete Manager

There is a thread in the QM dedicated to ensuring space freed by deleted urls is actually returned to the system.

Whenever a url is deleted from the LCC DB, any stored content (defined by a machine-id and filename) is required to be deleted. This deletion is performed by communicating with the BOT-M, sending a "delete" command, along with the machine-id and filename pair. The BOT-M passes this to the appropriate BOT-M on the target machine which performs the file deletion and adds the filename to its pool of filenames available to be re-used.

However, if the BOT-M on the target machine is not running (or indeed, the target machine itself is not running) it is not possible to perform the file deletion. The task of the delete-manager thread is to manage this eventuality.

It passes the machine-id+filename pairs to the BOT-M for deletion, one at a time. In the case of an error-return, indicating a BOT-M is not running for the requested machine, the machine-id+filename pair is saved to be deleted later, when the appropriate BOT-M comes on line. Note that this state must be saved to a file or external db (labelled fsmdb in the general architecture diagram) so that the information is not lost over system restarts.

The delete-manager will either try periodically to delete machine-id+filename pairs previously found to be in error or, depending on the system structure, might be able to be notified directly when a BOT-M comes on line serving a particular machine-id.

Some Notes

One important point is that after a fetch completes the QM will delete the appropriate record from the QDB. Just before this deletion it will examine, one last time, the operation field of that record -- if it has changed to "nothing" it implies that a delete operation has been performed *during the actual fetch* of that url, and the QM will then perform a delete.

The BOT-M process can have many robots behind it, and can perform many url fetches in parallel. It will report the number of simultaneous fetches it can actually perform, and the QM will modify this so as to slightly overload the BOT-M to reduce latency in getting urls to fetch. This value is reported by the BOT-M to the QM when the QM connects to the BOT-M and may, in fact, be subsequently updated by the BOT-M if it notices robots coming and going.

The QM will feed the BOT-M process urls until there are this many fetches outstanding. It will then only submit another url to the BOT-M when the BOT-M gives the results of a previous fetch request.

It should be noted that there is a wealth of implementation detail regarding the general implementation of the QM, and many of these details have not been discussed here. Hopefully, however, there are enough guidelines listed here to give an indication of how the system is intended to work, and that it should work as intended.

QM Graceful Shutdown

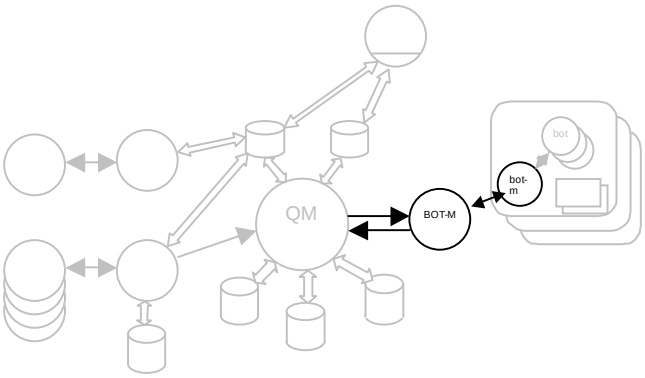
There can be a number of reasons why the QM cannot keep up with the incoming update notification load:

- the LCC DB interaction cannot keep up.
- the robots cannot keep up.
- other queue-related processing cannot keep up.

The QM recognises any of these situations by the fact of having the QMDB or other queue structures grow beyond configured maximum sizes. In this case it prevents further url submissions from UNSes by simply closing down its listening socket. Individual UNSes will notice this and temporarily buffer any other incoming url sets from CPs until they, too, reach limits, at which time they will refuse further CP connections.

After the QM has processed enough of the queue to allow normal processing of incoming url submissions to resume, it will once again start listening for connections on its listen socket. Individual UNSes will notice this, and start once again forwarding update notifications.

7. BOT-M: The Robot Multiplexer



Overall Mission

A BOT-M process performs the basic multiplexing between a single source of urls-to-fetch (the QM or another BOT-M) and multiple fetchers. These fetchers can either be a set of robots, or a set of BOT-Ms talking to robots. The more complicated case of talking-to-robots is discussed here, with the talking-to-multiplexers case discussed in a later subsection.

When the BOT-M process is talking to a set of robots, it is the BOT-M that generates the (machine-local) filename for the robot to store into. Ie, it is the BOT-M that performs filename management. It is intended that there is a BOT-M process for each machine that has a collection of robots.

A BOT-M process is connected to by each robot. As such, the BOT-M can adapt to changing numbers of robots.

A BOT-M makes a connection to its url-source (either the QM or the central BOT-M). If the url-source is restarted, the BOT-M will reconnect to it.

The url-source passes url-fetch requests to the BOT-M, which gives it to the next free available robot. Once the robot has fetched the url a success or failure indication is returned to the BOT-M for that url, which in turn passes the result to the url-source along with information as to where the url content was stored.

Some More Details

The BOT-M keeps internal state regarding:

- Filename management of files for robots to store into.
- For each robot, a TCP connection and how many urls the robot can fetch in parallel (more than one if the robot is threaded).
- For each actively fetching robot, what urls it is fetching and where it is fetching into (information only required in case the robot is killed before fetching completes).
- A set of idle robots.
- The (small) sequence of urls submitted by the source that have not been given to a robot yet.

A BOT-M is intended to be running on the same machine as a bunch of robots. It is configured with a directory into which url content is stored, and will generate subdirectories and filenames automatically within that directory to be stored into by individual robots. This directory must be accessable to both the BOT-M and robots of course, and is intended to be on the local machine.

The only permanent state a BOT-M works with is to do with filename management -- an indication of the next new filename to generate, and a table of free filenames. It preserves no other changing information over restarts.

When it connects to its url-source, the BOT-M passes an indication of how many robots it has connected. Whenever robots connect and disconnect from the BOT-M, the BOT-M will report a new figure to the url-source. This guides the url-source as to how many outstanding urls can be passed to the BOT-M. (Note that the url-source will likely overload the BOT-M to a certain extent to ensure the robots are as busy as possible.)

When a url-fetch notification is received from the url-source, it is placed into an in-memory queue in the BOT-M. When a robot becomes free, it is given the url to fetch along with a filename to store the content into (either generated for a new filename, or a re-used old filename).

When a url-delete notification is received from the url-source, the physical file is deleted directly by the BOT-M, and the filename is placed on the filename freelist to be re-used later as the target of another url fetch. In a normally running LCC, this list of filenames is unlikely to grow very large. In fact, it should have very close to zero entries normally.

If the BOT-M notices that a robot has been killed that was serving a url fetch, and that had not yet provided a response, the url that was being fetched is given to another free robot with the same destination filename as before.

When a robot reports success, the BOT-M forwards this information to its url-source. If a robot reports failure, the destination filename that was targeted is placed onto the free filename list and the BOT-M communicates the failure to its url-source.

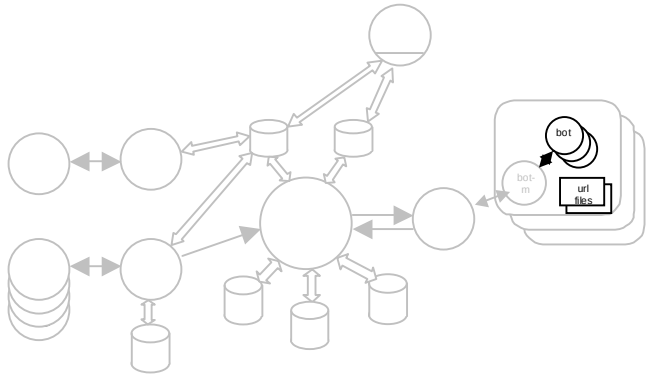
If a BOT-M is killed during processing and restarted, it is up to its url source to resubmit urls lost during processing.

When Talking to Other BOT-Ms

In the case where a BOT-M is talking to other BOT-Ms (as is the case in the general architecture diagram for the BOT-M connected to the QM), the BOT-M:

- Does not perform any filename management tasks. It just forwards filename information it receives.
- Realizes that each connection it has to another BOT-M can have more than just a single url fetch request outstanding (as is possible with BOTs, in fact) and that the BOT-M can possibly be "overloaded" slightly if that is required to improve throughput. The amount of "overloading" will be configurable.
- Tags each BOT-M connection with the machine-id of the machine that is being served. When a url-delete command is received (a machine-id+filename pair) it then knows to which BOT-M the command should be forwarded.

8. BOT: URL Fetching Robot



Overall Mission

A robot in the LCC is given a port to connect to from which to receive requests; this port is the robotting port owned by the BOT-M normally present on the local machine. The robot will communicate how many urls it can fetch in parallel after it connects (a non-threaded robot can fetch one url at a time, a multi-threaded robot can fetch as many urls as it has threads).

For flexibility a robot may instead be asked to listen on a socket for an incoming connection instead. This will mainly be used as a debugging aid where the robot can be telnetted to, for example, and XML typed in to perform a fetch.

Once the connection is established, the robot expects data to be sent down the conenction giving the name of a URL to fetch, an optional timestamp, optional expected-mime-type, and the name of a file to write the content to.

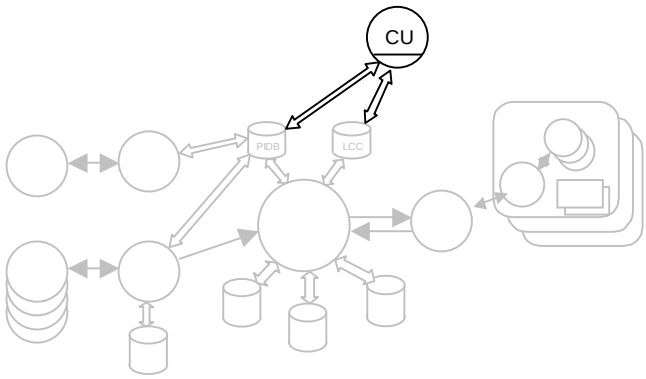
It attempts to fetch the url, placing its content into the nominated file, and returns a result indicating:

- success/failure.
- md5 checksum (possibly optional).
- file size.
- reported last-file-modification timestamp.
- reported mime-type.

When a robot is threaded, each internal thread is nearly completely independant, and acts as a self-contained robot. The only thing shared is the TCP connection down which requests arrive and responses are sent -- this is managed with a simple low-overhead mutex.

Some implementation details such as mime-type mismatches, file-size and fetch-time limits are not discussed here.

9. CU: A Content User



A content user is effectively able to perform a select on the LCC DB having row-information returned, or row+file-content information. This information is packaged up in different ways, depending on the CU. Eg, a .tar.gz file automatically being made of all content matching a "changed after x" query.

CAPI: The C-API

For content users to use the system, a C-API is provided. With direct access to the underlying LCC DB and PI DB, this API allows selection of urls of interest.

It is possible to retrieve url content with the API as it also communicates with the BOT-M of each machine. It is likely to find the ports of these BOT-Ms by communicating with the central well-known one connected to the QM.

Other APIs serving different (more dedicated) needs are intended to be created on the top of the C-API. For example:

- An "XML" API, where the query is sent to a socket as an XML stream, and the results are received as an XML stream.
- A ".tar.gz" API, where the results are automatically packaged into .tar.gz form.

In the case where the LCC is multiplexed internally it is the job of the C-API to present a single unified view to the CU. It does this by performing the query and returning the results from each LCC DB in turn.

10. LCC Graceful Shutdown

This section repeats information found in the "UNS" and "QM" sections showing how the LCC copes with a fetch load that is too high.

An "overload" condition (too many outstanding urls to fetch) is noticed by the QM. There can be a number of reasons why the QM cannot keep up with the incoming update notification load:

- The LCC DB interaction cannot keep up.
- The robots cannot keep up or some robots are not functioning.
- Other queue-related processing cannot keep up.

The QM recognises any of these situations by the fact of having the QDB or other queue structures grow beyond configured maximum sizes. In this case it prevents further url submissions from UNSes by simply closing down its listening socket.

Individual UNSes will notice this and temporarily buffer any other incoming url sets from CPs until they, too, reach limits, at which time they will refuse further CP connections giving a "try again later" error indication.

After the QM has processed enough of the queue to allow normal processing of incoming url submissions to resume, it will once again start listening for connections on its listen socket. Individual UNSes will eventually notice this, and start once again forwarding update notifications, initially from any buffered sets.

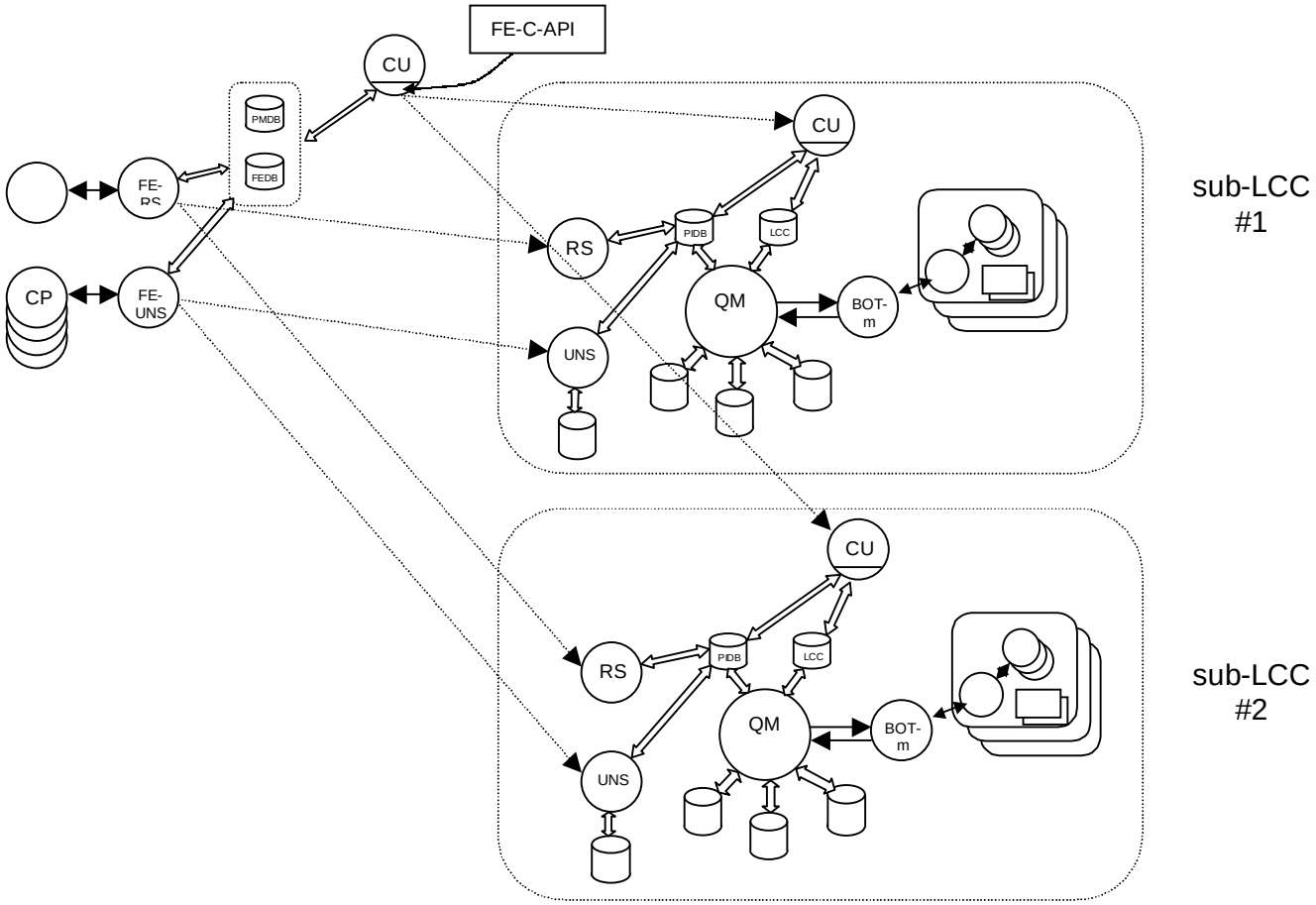
11. LCC Reliability Notes

Some notes related to the general reliability of the system are presented here.

When url notifications are presented to the system, it is desired that it is not "lost" without an error indication eventually being given to the provider. This is achieved in the following ways:

- If the set of urls cannot be stored because of an overload condition, the CP is notified immediately by the UNS it has connected to.
- When the set of urls is passed from the UNS to the QM, the UNS only provides a positive response to the CP only after the QM provides a positive response. The QM provides a positive response only after the url information has been absorbed into queue related structures that are on disk (ie, the QDB and QPSDB).
  If the QM provides a negative response, the UNS will forward this to the CP.
  If the QM "goes away" (either dies, is killed, or closes its connections because of overload) the UNS will provide a positive response to the CP if the url set can be written to the unsdb backup file, otherwise a negative response will be forwarded to the CP.
- If there are ways where the parts of the LCC can crash that would otherwise result in "information loss", ie urls being silently dropped, a "cache crash" flag will be set. The value of this flag will be reported to CPs when they connect, and they can take it as an indication that they should provide the "full set" of their urls to the system rather than simply update notifications.

12. LCC Multiplexing



When the LCC grows beyond a certain size there may be performance problems. These problems could come from one of a number of areas (eg, disk space, database size and access, memory limits). It is intended that it be possible to "split up" such a large LCC into multiple smaller LCCs.

For ease-of-splitting, any particular provider will be deemed to be in a single sub-LCC. This mapping of provider-id to sub-LCC will be stored in a dedicated db, labelled “PMDB” (Provider Mapping Database) above. Each particular sub-LCC will have its vital statistics recorded in some central location, named here as the “FEDB” (Front End Database); this indicates how to communicate with that sub-LCC. To make a multiplexed LCC be transparent to users, versions of the interface processes (the RS, CP and C-API) will have to be developed that look at these databases where necessary to discover which sub-LCC to communicate with and how to communicate with it. These processes are labelled FERS, FECP and FEC-API above.

The UNS, for example, will receive a connection from a particular provider id which will be mapped to a particular sub-LCC, the one containing that provider. The UNS will then forward what it receives from its connection directly to the appropriate port, and return any results. Ie, it simply turns into a forwarding process with the UNS “underneath” (the UNS inside the sub-LCC) doing the real and, by now, distributed work.

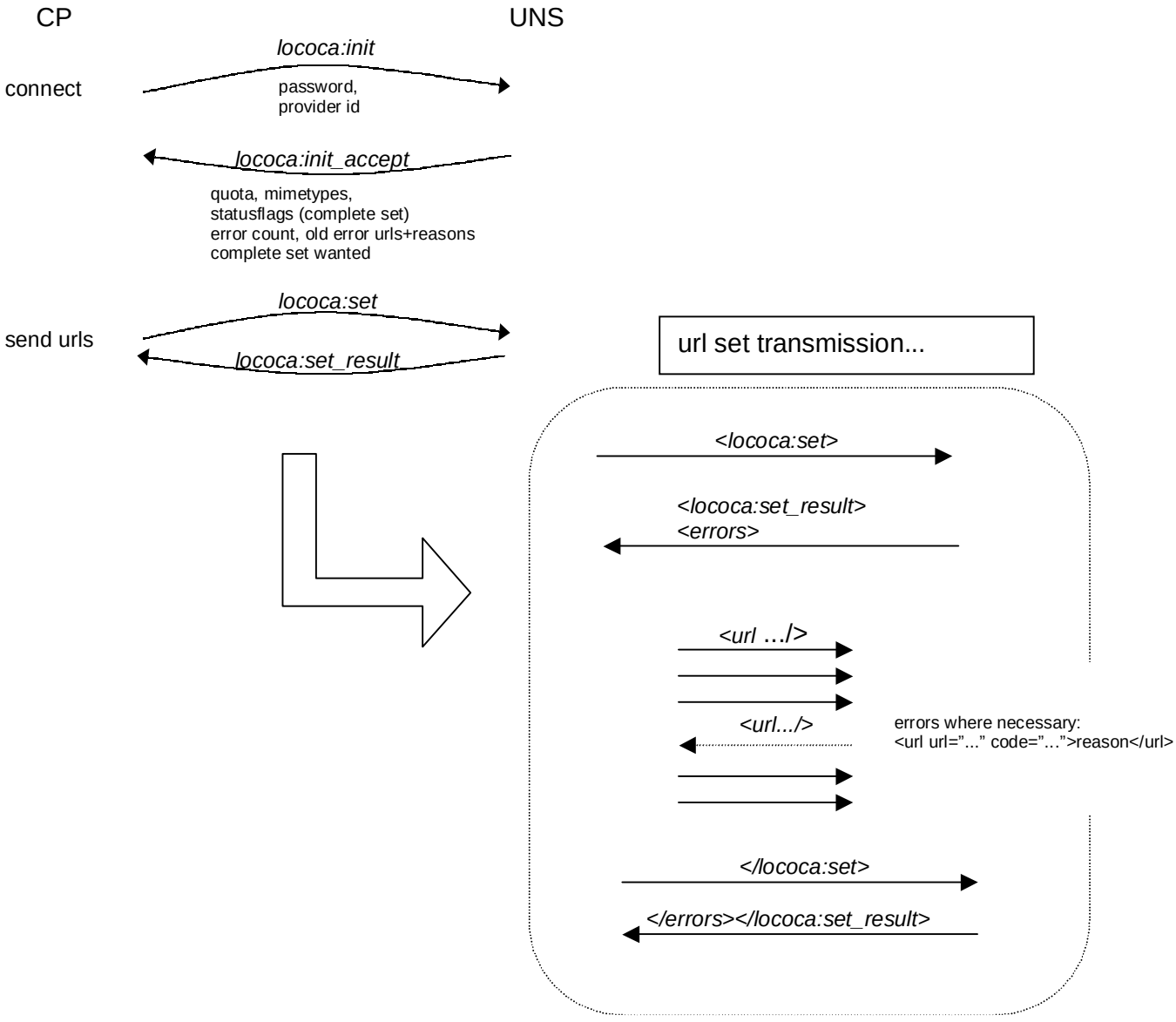
From the CU end, a given CU still sees a single LCC. The C-API manages the fact that there are sub-LCCs and is responsible for forwarding the CU-request to each sub-LCC and concatenating their individual results. It discovers the sub-LCCs available by looking in the FEDB. Depending on the implementation of the C-API, it can either interrogate the sub-LCC LCC DB's in-turn directly, or it may use an XML-API (for example) to communicate with each sub-LCC in turn. The diagram shows it communicating with a CU (eg, an XML-API).

When the time comes to split or re-split a given LCC there will be various procedures to follow, and tools to aid the transition. These are not discussed in this document.

Splitting of an LCC into sub-LCCs will not be possible with the initial implementation.

13. CP/UNS Protocol

The information transfer between a CP and the UNS appears in the following diagram:



Communication between the Content Provider (CP) and the Update Notification Server (UNS) takes place using messages in XML-format. Each request, and the reply to that request, are in the form of a contained XML, i.e. the end-tag of the first tag received signals the end of the message or the reply.

No encryption of the messages between the CP and UNS will be implemented yet. Initially the interaction will be in plain-text, i.e. visible to outside snoopers. This may be acceptable for the initial implementation. Future implementations will need to address this issue.

Connection Initiation

The connection between the CP and the UNS is initiated by the CP by attempting to establish a TCP-connection with the UNS on a designated port. After the connection is established, the CP is expected to send:

```
<lococa:init xmlns:lococa="http://www.lococa.org/1.0">
   <provider id="providerid" passwd="password"/>
</lococa:init>
```

The "id" attribute contains the ProviderID that has been issued by the Registration Server (RS) when the CP registered. The "passwd" attribute contains the password that is needed to gain access to the system: this was also issued by the RS.

Connection Failure

If the provider/password combination is incorrect, or any other error condition exists at the UNS (such as an internal overload), the UNS responds with:

```
<lococa:init_rejected>
   <reason code="CC">error message</reason>
</lococa:init_rejected>
```

The “reason” container gives an indication of the error. The "code" attribute contains an error code identifying the type of error. This code is language independent and can be used to translate the error message into languages different from English. The error message consists of an English error message explaining the reason for the error.

Please note that the XML-namespace definition is missing from the reply container. Since the UNS has received the namespace definition from the CP, the UNS will answer using the same namespace definition and thus it does not need to be sent along as well.

Connection Success

If access was granted to the CP by the UNS, the reply of the UNS has the following form:

```
<lococa:init_accepted>
   <connect_info seq="NN" lastConnectIP="1.2.3.4"/>
   <mime>text/*</mime>
   <mime>application/pdf</mime>
   <mime>application/ms-word</mime>
   <quota>
      <files used="NN" free="NN"/>
      <space used="BB" free="BB"/>
      <fullset allowed="NN" wanted="yes|no"/>
   </quota>
   <processing_status errors="XX" processing="YY">
      <errors>
         <url code="CC" url="http://www.server.com/file.html"...>Not Found</url>
```

```
        <url code="CC" url="http://www.server.com/file2.html"...>Not Found</url>
        ...
      </errors>
    </processing_status>
  </lococa:init_accepted>
```

The <*connect_info*> container gives an incrementing sequence number (NN) that the provider can remember if desired to ensure no-one else has connected as this CP. The "lastConnectIP" attribute specifies the IP-number from which a successful connection was made the last time.

The <*mime*> containers contain the mime-types that the LCC will allow. The <*quota*> container contains the information about quota that apply to the provider. The <*files*> container pertains to numbers of files allowed. The <*space*> container pertains to number of bytes of diskspace allowed. The <*fullset*> container indicates the number of full set requests that are still allowed and whether the LCC would like to receive a full set specification from the CP.

The <*processing_status*> container gives an indication of how many urls supplied by this provider are still in the process of being fetched and, also, the total number of fetch errors that have been enountered for this provider since the last CP connection. Each <*url*> container inside the <*error*> container identifes a url that could not be fetched. Each <*url*> container contains the error code and an english language error message, the URL that could not be fetched, and all other information related to this url passed by the CP. The LCC can store a limited number of errors of this form, the number being a configuration parameter. The "errors" attribute to <*processing_status*>, however, is the total number of errors encountered.

The <*fullset*> container indicates two things:

- How many "full sets" are still allowed to be uploaded by this CP to the LCC. There is a configurable number of full sets that are allowed to be uploaded by any particular CP within a given time period (the normal mode of operation is to have update sets being submitted by the CP).
- Whether the LCC explicitly wants the CP to provide a "full set". This can occur for a number of reasons and the state is "sticky". Ie, if the CP decides not to send a full set now, but a modification set, the next time the CP connects to the LCC the "wanted" attribute will still be set to "yes".

Update Set

When the CP is ready to send either a full set or partial set of URLs, it is expected to send the following container:

```
<lococa:set set="full|partial" urlprefix="http://www.server.com/">
  <url c="index.htm" b="frameset.htm?index.htm" f="index.htm?lococa".../>
  <url c="page1.htm" b="frameset.htm?page1.htm" f="page1.htm?lococa".../>
  ...
</lococa:set>
```

The "set" attribute indicates whether the CP is providing a full set or a partial set. The "urlprefix" attribute specifies the string that should be prefixed before all URLs that do not start with a protocol specification.

Each <*url*> container contains the specification of a single conceptual URL. The "c" attribute indicates the "conceptual URL", the "b" attribute indicates the "browse URL", the "f" attribute specifies the "fetch

URL". There will also be a modification time indication, file size indication and md5-checksum provided here.

Update Result

During the receiving of the <*lococa:set*> container, the UNS starts sending back the <*lococa:set_result*> container, which has the following form:

```
<lococa:set_result>
   <errors>
      <url code="CC" url="http://www.server.com/index.html"...>Not allowed</url>
      ...
   </errors>
   <set_accepted received="NN"/>
      or
   <set_rejected code="CC">reason</set_rejected>
</lococa:set_result>
```

If any urls are found to be in error (due to syntax, mime-type or other easily discoverable reasons by the UNS) these urls are returned in appropriate <*url*> containers. If no urls are found to be in error, the <*errors*> container is either missing or empty.

Once the UNS has encountered the <*lococa:set*> tag, it will generate a <*set_accepted*> or <*set_rejected*> tag. When <*set_accepted*> is seen by the CP, it indicates a complete, successful transaction between the CP and the UNS, and a count of valid urls is returned.

Both the UNS and CP will then close their connections.

In an asynchronous environment, the receipt of the entire <*lococa:set_result*> container could happen before the sending of the entire <*lococa:set*> container by the CP is completed, particularly in the case when the whole set is rejected. The CP is not obliged to finish the <*lococa:set*> container in that case.