Distributed Cache Manager and API

Jim Challenger

Snail IBM T. J. Watson Research Center
30 Sawmill River Road.
Hawthorne, NY 10532

email challngr@watson.ibm.com

Phone Tieline 863-7175
external 914-784-7175

Fax 914-784-7595

Arun Iyengar

Snail IBM T. J. Watson Research Center
30 Sawmill River Road.
Hawthorne, NY 10532

email aruni@watson.ibm.com

Phone Tieline 863-6468
external 914-784-6468

Fax 914-784-7595

Contents

Overview

The cache manager provides multi-threaded management of multiple caches within a single process. Each cache is completely independent of the others and may be configured to enforce differing policies.

Although completely general in nature, the cache manager has been designed with the intention of managing cached, dynamically generated HTML pages for WWW sites.

A cached object such as an HTML page could have been constructed from several other data aggregates such as database tables. The cache manager allows these relationships to be specified. Each cached item may have a list of *dependencies* associated with it. For example, an item on a dependency list for a cached HTML page might represent a database table whose value affects the contents of the page. The cache provides an API function for specifying that the database table has been updated. This causes all cached objects which depend on the table to be invalidated. Using this mechanism, a routine which performs updates to a database can update the cache without knowing the specific URL's which are affected by the update.

The cache manager is designed to work in a networked environment in which a single cache is shared among multiple processes on multiple machines.

An API is provided to enable applications to communicate with the cache manager. Functions to manage tokens, data, and perform cache administration are provided.

Building the cache manager and API

Getting source

Current source for the Cache Manager is freely available but we must track distribution. Please contact Jim Challenger at challngr@watson.ibm.com. Current source status is:

- Source - AIX 4.1.4 (compressed tar, 2.5M)
- Source - Windows NT 4.0 (in progress, available 3Q 1997)

Distributions include all source, makefiles, and tools required for building.

Build Prerequisites

To build the cache manager as distributed the following software is reqired:

- GNU make.
- Flex, from the Free Software Foundation.
- AIX 4.1.4 or AIX 3.2.5
- Optionally, Bison, fromt the Free Software Foundation

The makefiles are constructed for gmake, from the Free Software Foundatation. It is straightforward to convert them to work with IBM make, but converted makefiles are not part of this distribution.

A copy of gmake and flex are provided in the source directory with the cache manager for convenience.

Compiling

The cache manager consists of three compoents:

1. A thread package, providing operating system independence,
2. A transaction framework, called the ‘‘daemon framework’’, providing network independence, and
3. the cache manager itself.

The source is organized into three directories, thread, daemon, and cachemgr. There are several build options available:

1. **Default.** This uses a cooperative threading model and the GNU version of yacc, ‘‘bison’’. This version is not appropriate for product-level work because of the use of bison; however, bison provides better diagnostics than most versions of yacc, and is recommended for development work.

   To build this version, cd to the top directory, and run

   ```
   gmake
   ```

2. **Use a cooperative thread model, using operating system supplied yacc**. To build this version, cd to the top directory and run

   ```
   gmake use_yacc
   ```

3. **Use a preemptive thread model, with bison**. This uses the pthread interface.

   ```
   gmake pthread
   ```

4. **Use a preemptive thread model, with yacc:**

   ```
    gmake pthread_yacc
   ```

When the build is complete, the cachemgr directory will contain several files:

- cachemgrd - the cache manager
- cache_api.shr.o - an AIX shared library with the API calls documented below.
- libcache.a - an AIX (nonshared) library with the API calls documented below
- test_cached - a test program to verify the build and basic function.

Installing

To install the cache manager, perform the following steps:

- Create a directory on a file system large enough to contain the cache manager executables and logs. A minimum of ten megabytes is recommended.
- Copy the files created during the build process to your new directory.
- Create a configuration file.

- Create a directory for each cache named in the configuration file.

Installation is complete. To start the cache manager, issue the command,

```
cachemgrd -c <config_file>
```

where config_file is the name of your configuration file. Verify successful startup by checking the cache manager log.

Verifying

To verify installation,

1. Create a small file (which can be overwritten) in some directory,
2. Configure two caches called test1 and test2,
3. (re)Start the cache manager.
4. Verify that the caches were correctly configured and were started by checking the cache manager log.
5. Run the command

   ```
   test_cached -f1 <filename>
   ```

   where filename is the name of the file you created.

Several messages will be printed by the verification routine, some of which are deliberate error messages. Verification is successful if the last message is

```
cache verification was successful.
```

The test_cached program accepts several parameters:

```
test_cached [-h hostname] [-p port] [-f1 fileid] [-c1 cacheid] [-c2 cacheid] [-to
```

where

-h	is the hostname where the cache is running, if different from where test_cached is run.
-p	is the port the cache is configured on, if different from the default (7175).
-f1	is the name of a small file to be used to verify the file APIs.
-c1	is an alternate cache id to be used for testing other than ‘‘test1’’.
-c2	is an alternate cache id to be used for testing other than ‘‘test2’’.
-to	is the communications timout to be used, other than the default of 30 seconds.

Configuring the Cache Manager

Concepts

The configuration file consists of a series of named stanzas. One stanza per cache must be defined, as well as one stanza for the cache manager itself.

A stanza consists of a label followed by a series of statements, one per line, enclosed in braces. Example:

```
cache0 {
    caching = on
    mem-size = 10MB
}
```

The cache-manager stanza

The cache-manager stanza defines the parameters of the manager itself: network information, logging status, tracing status. This stanza is required and must be labeled cache-manager.

Keywords accepted in the cache-manager stanza are:

log

This defines the name of the file used for cache manager logging. This is optional and if not specified, messages are written to the console. This log shows activity for all transactions for all caches and is intended primarily for debugging and problem analysis. For logging on a per-cache basis, use tran_log. Example:

```
log=/u/cached/logs/cached.log
```

port

This specifies the TCP/IP port which is listened to by the cache manager for incoming requests. This is optional, and if not specified, is determined by (a) checking /etc/services for the name ‘‘ibm-cachemgrd’’; if not found, (b) use the default port 7175. Example:

```
port=7177
```

connection-timeout

specifies the maximum length of time in seconds the cache manager should allow a pending read to be left active. If this time is exceeded the connection is dropped. This is optional, and defaults to 30 seconds. Example:

```
 connection-timeout=60
```

logging

Specifies whether logging should be done. It should be specified as either ‘‘yes’’ or ‘‘on’’ to indicate logging is required, or as ‘‘no’’ or ‘‘off’’ to specify logging should not be performed. This is optional, and if not specified, defaults to ‘‘no’’. Example:

```
 logging=yes
```

wrap-log	Specifies whether the log should be wrapped. This should be specified as yes or no. This is optional, and if not specified, defaults to no. If specified as yes, the current log is closed when it reaches its maximum size (see log-size, below), has .old appended to its name, and a new log is opened. Only one generation of log is maintained (that is, existing .old files are overwritten). Example: `wrap-log=yes`
log-size.	Specifies the maximum size in bytes to which a log is allowed to grow, if wrap-log is specified. This is optional, and if not specified, defaults to 64000. Example: `log-size=100k`
trace-flags	This specifies the level of messages to be written to the log. This is optional and if not specified, only cache manager startup and shutdown messages are logged. Example: `trace-flags= D_OPEN D_READ D_FRAMEWORK`

Trace Flag Definitions

Trace flags are intended for diagnostic use and can provide considerable information about the inner workings of the cache manager.

Trace flags should be specified as a blank-delimited list. Supported flags are:

D_ALL	Enable all trace flags.
D_ADMIN	Log administrative command execution.
D_ALLOCATE	Log storage allocation info.
D_API	(Not used yet)
D_CLEAR	Log cache clear operations.
D_CLOSE	Log cache (datum) close operations.
D_FRAMEWORK	Log operations specific to the underlying network framework layer.
D_MANAGER	Log operations specific to the cache manager as opposed to any specific cache.
D_OPEN	Log cache (datum) opens.
D_PURGE	Log cache purge operations
D_READ	Log cache read operations.
D_STATISTICS	(Currently unused)

D_TIMING	Enable timing of events.
D_TRANSACTION	Enable messages relating to the transaction mechanism.
D_WRITE	Log cache write operations.

Cache Definition Stanzas

Each cache which is managed by the cache manager must be defined in the configuration file. Valid keywords include:

caching

specifies whether this cache is to be enabled when the cache manager starts. It should be specified as yes or no. The default is yes. If set to no, the cache is defined in the cache manager but not activated. It may be activated later via the cacheadm command. Example:

```
caching=yes
```

fs-size

specifies the maximum space that may be used in the filesystem by objects in this cache. When exceeded, the cache manager will delete sufficient items, starting with the oldest, to bring total space occupied by the cache within bounds. You may effectively disable automatic purging of entries by setting this to a large number; however, if the physical filesystem space is exceeded, attempts to add new entries to cache will fail. Units may be specified as nnB for nn bytes, nnKB for kilobytes, or nnM for megabytes. The default is fs-size=0 (no caching to disk). Example:

```
fssize=64MB
```

mem-size

specifies the maximum amount of memory that may be used by all of the objects in this cache. When exceeded, the cache manager will delete sufficient items, starting wit the oldest, bo bring total space occupied by the cach within bounds. You may effectively disable automatic purging of entries by setting this to a large number; however, if the cachemgrd process consumes too much space the operating system may terminate it. Units may be specified as nnB for nn bytes, nnKB for kilobytes, or nnMB for megabytes. The default is 1MB. Example:

```
mem-size=1000KB
```

lifetime

specifies the maximum length of time an item may be held in cache. When exceeded, the item is marked expired. The item is not deleted from cache unless the fssize (if it is cached on disk) or memsize (if it is cached in memory) limits are reached. Items marked expired are deleted before all other items if memsize or fssize limits are reached. Lifetime checking may be disabled with the check_expiration keyword. Units may be specified as nnS for nn seconds, nnM for minutes, or nnH for hours. The default is 5 minutes. Example:

```
lifetime=600S
```

check-expiration specifies whether lifetime checking should be performed. The default is yes. If set to no items are never marked expired. The default is 60 seconds. Example:

```
check-expiration=no
```

datum-memory-limit specifies the maximum size a specific data item may occupy within the memory cache. If an item is too large for memory, the file cache is checked, and if it will fit there, is placed in the filesystem instead. If it does not fit in the filesystem, the attempt to cache the item will fail. If the item is smaller than datum_memory_limit but no room exists in the memory cache, the oldest items are deleted from the memory cache to accommodate the new item. Units may be expressed in bytes(B), kilobytes(KB), or megabytes(MB). The default is 1KB. Example:

```
datum-memory-limit=200KB
```

datum-disk-limit specifies the maximum space in the file cache an item may occupy. No object larger than this will be accepted for caching. If the object is smaller than datum_disk_limit but no space remains in the file cache, the oldest items are deleted to accommodate the new item. Units may be expressed in bytes(B), kilobytes(KB), or megabytes(MB). The default is -1 (no limit). Example:

```
datum-disk-limit=1MB
```

stat-interval specifies the time between creation of statistics records. If set to zero, no statistics records are written. Units may be nnS for nn seconds, nnM for minutes, or nnH for hours. The default is 0 (don’t collect statistics). Example:

```
stat-interval=1M
```

reset-stat-counters specifies whether or not statistics counters should be reset to 0 each time they are written to the log file. The default is yes. Example:

```
reset-stat-counters=no
```

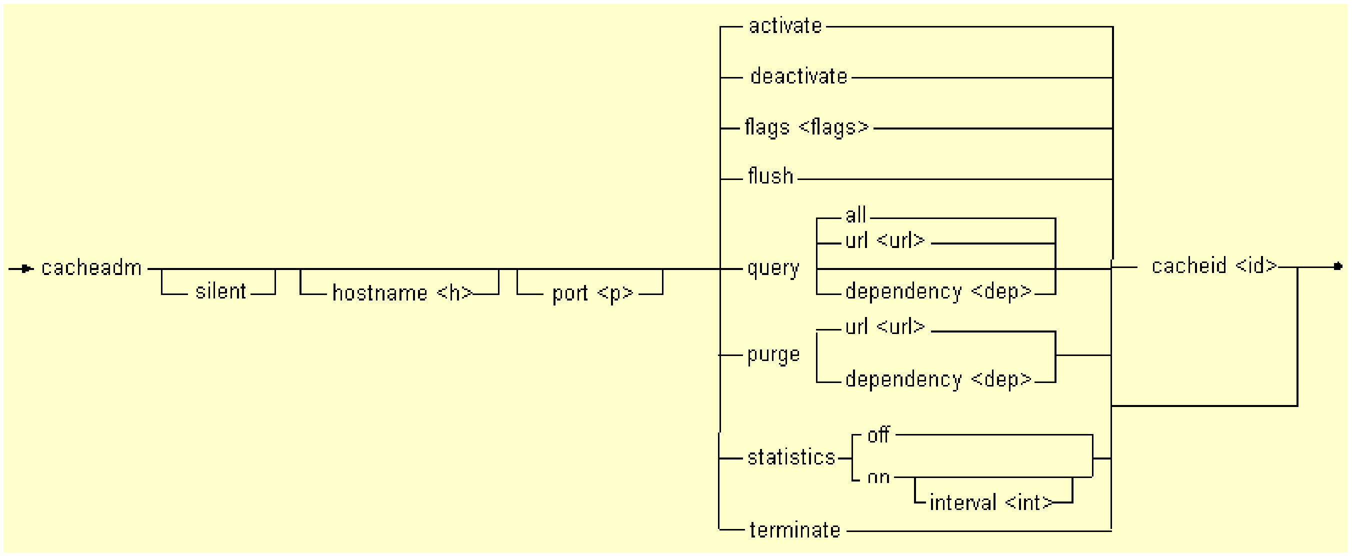
root specifies the name of the directory to hold the cache items. On startup, the filesystem containing this directory must be at least as large as fssize or the cache will not be started. root may be specified as a path relative to that in which the cache manager was started, or an absolute path. This parameter is required. Example:

```
root=/u/cached/caches/cache0
```

stat-file	specifies the name of the file to be used for logging statistics for this cache. This parameter is required if stat-interval > 0. Example: `stat-file=/u/cached/logs/cache0.stats`
hashing	specifies whether dependency structures should be maintained for this cache. If not specified, the default is *off*. Example: `hashing=on`
tran-log	This defines the name of the file used for transaction logging on a strictly per-cache basis. Cache transaction log files are separate from cache manager log files which are used to log overall cache manager activity. This parameter is optional and if not specified, a transaction log for the cache is not created. Example: `tran-log=/u/cached/logs/cache0.log`
tran-logging	Specifies whether transaction logging for the cache should be turned on when the cache manager first starts up. This parameter is ignored unless a valid transaction log file is specified via the tran-log parameter. It should be specified as either ‘‘yes’’ or ‘‘on’’ to indicate logging is required, or as ‘‘no’’ or ‘‘off’’ to specify logging should not be performed. The parameter is optional, and if not specified, defaults to ‘‘no’’. Transaction logging can be turned on or off while the cache manager daemon is running via the cacheadm command (provided a valid tran-log parameter was specified in the configuration file). Example: `tran-logging=yes`
wrap-tran-log	Specifies whether the transaction log should be wrapped. This should be specified as yes or no. This is optional, and if not specified, defaults to yes. If specified as yes, the current log is closed when it reaches its maximum size (see tran-log-size, below), has .old appended to its name, and a new log is opened. Only one generation of log is maintained (that is, existing .old files are overwritten). Example: `wrap-tran-log=yes`
tran-log-size.	Specifies the maximum size in bytes to which a transaction log is allowed to grow, if wrap-tran-log is specified. This is optional, and if not specified, defaults to 64000. Example: `tran-log-size=100k`

The cacheadm Command

Use the cacheadm command to stop the cache manager, flush a specific cache, query a specific cache, enable or disable logging and logging flags, and start and stop statistics gathering. All parameters may be abbreviated to the minimum unique set of characters. The syntax of the cacheadm command is



where

activate	means ‘‘activate the specified cache’’. If the cache is already active, nothing is done. If a cache transaction log is being maintained, a transaction record in the following format is generated: `<date> <time> cacheadm_activate`
deactivate	means ‘‘deactivate the specified cache’’. If the cache is already inactive nothing is done. All pending operations are completed and no new ones accepted. When the last operation is complete, the cache is marked inactive. If a cache transaction log is being maintained, a transaction record in the following format is generated: `<date> <time> cacheadm_deactivate`
cacheid	specifies which of the managed caches the command pertains to (for relevant commands).
flags	specifies the state of the named flags should be toggled. The flags are specified as a blank delimited list of flags as described below.
flush	means flush the cache.

This command invokes the Cache API functions CacheInit, CacheClear, and CacheTerm. If a cache transaction log is being maintained, transaction records from these API calls are generated.

hostname specifies the host where the cache is running, if different from the machine where cacheadm is issued.

port specifies the cache port, if different from the default (7175).

purge specifies that a specific item should be purged from cache. If *url* is specified, the item with a key matching **url** is purged. This is accomplished by invoking the API functions CacheInit, CacheInvalidateRecord, and CacheTerm. If *dependency* is specified, all items with the associated **dependency** are purged and their keys written to stdout. This is accomplished by invoking the API functions CacheInit, CachePurge, and CacheTerm.

If a cache transaction log is being maintained, transaction records from the API calls are generated.

query returns data about a cache, if only the cacheid is specified. Query returns the list of cache keys associated with the **dependency list** if *depenendency* is provided. This is accomplished by invoking the API calls CacheInit, CacheQueryDependentKeys, and CacheTerm. Query returns information about a **specific** cached item if *url* is specified. This is accomplished by invoking the API calls CacheInit, CacheQueryKey, and CacheTerm. Query returns information about **all** items if *all* is specified. This is accomplished by invoking the API calls CacheInit, CacheQueryAll, and CacheTerm.

The *all* options is intended for use by other programs which will format or interpret the results. Each line contains the following information: the item key, the item age, the item length, the item creation date, the item expiration date, and the date the item was last referenced. All dates are in standard Unix integer time format.

"cache query all" is very expensive and should be used sparingly.

If a cache transaction log is being maintained, the exact format of transaction records depend upon the parameters passed. If only the cacheid is specified, a transaction record in the following format is generated:

```
<date> <time> cacheadm_query
```

Otherwise, transaction records from the API calls which are invoked are generated.

silent instructs cacheadm not to issue any messages to the console (other than fatal error messages).

statistics enables and disables logging of statistics gathering for a specific cache. The cacheid parameter is required with this option. If an *interval* is specified with **statistics on** the interval between updates is set or reset to the specified number of seconds.

If a cache transaction log is being maintained, enabling logging of statistics results in a transaction record in the following format:

```
<date> <time> cacheadm_stat_activate
```

Disabling logging of statistics results in a transaction record in the following format:

```
<date> <time> cacheadm_stat_deactivate
```

terminate	means terminate the cache manager.
tranlogging	enables and disables transaction logging for a specific cache. The cacheid parameter is required with this option. This command will only have an effect if a valid transaction log for the cache was specified in the configuration file via the tran-log parameter.

Enabling transaction logging results in a transaction record in the following format:

```
<date> <time> cacheadm_tranlogging_activate
```

Disabling transaction logging results in a transaction record in the following format:

```
<date> <time> cacheadm_tranlogging_deactivate
```

Cache Manager Log flags

The log flags described in this section are only applicable to cache manager logs and not to cache transaction logs. The log flags correspond directly with the trace flags which are set in the configuration file. This provides a way to dynamically modify the logging level without restarting the cache. Most log flags can be turned on by specifying the first character as a plus sign (+) or turned off by specifying a minus sign (-). For example:

```
    cacheadm -l +D_WRITE -D_TIMING
```

This example turns on tracing of all cache write activities, and turns off the transaction timing code.

Valid flags are:

D_ALL	Turn on all trace flags.
D_NONE	Turn off all trace flags.
[+|-]D_ADMIN	Turn on(+) or off(-) tracing of administration commands.
[+|-]D_ALLOCATE	Turn on(+) or off(-) tracing of storage allocation.
[+|-]D_CLOSE	Turn on(+) or off (-) tracing of object close commands.
[+|-]D_FRAMEWORK	Turn on(+) or off(-) tracing of low-level communication activities. This is usually used only for debugging by the cache developers as it can be difficult to interpret the messages without source listings.
[+|-]D_MANAGER	Turn on(+) or off (-) tracing of cache management functions.
[+|-]D_OPEN	Turn on(+) or off (-) tracing of object open commands.
[+|-]D_PURGE	Turn on(+) or off(-) tracing of object purge operations.

[+|-]D_READ	Turn on(+) or off(-) tracing of object read operations.
[+|-]D_TIMING	Turn on(+) or off (-) transaction timing messages.
[+|-]D_TRANSACTION	Turn on(+) or off(-) tracing of transaction management functions.
[+|-]D_WRITE	Turn on(+) or off(-) tracing of object write operations.

Interpreting the cache manager log

Log messages are of the form

```
date time action [flagid] (threadid) (cacheid) message
```

where

date	is the date of the message
time	is the time of the message
action	is a 1-work description of the type of action which caused the message to be emitted
flagid	is a 1-letter identification of the trace flag which caused the messgae to be emitted
threadid	is the id of the thread the action is occurring in
cacheid	identifies the cache the action is being performed for
message	is the message text

Flagids correspond to trace flags thus:

Code	Flag
A	D_ALWAYS
C	D_CLOSE
L	D_ALLOCATE
M	D_MANAGER
N	D_ADMIN
O	D_OPEN
P	D_PURGE
R	D_READ
S	D_STATISTICS
T	D_TRANSACTION
M	D_TIMING
W	D_WRITE

Statistics

Several statistics regarding internal operation are kept and optionally written to the accounting log. A sepparate log per cache may be maintained, or all statistics may be written to the same log. This log is configured with the statistics_log and statistics_interval keyword of the configuration file. Statistics gathering can be modified without stopping, reconfiguring, and restarting the cache manager with the cacheadm statistics command. Note however, that changes made via cacheadm statistics are not saved across restarts of the cache manager.

The statistics log is a plain ascii file suitable for processing or import by spreadsheets or database programs. Three types of records are written:

- Initialization records document the startup of statistics gathering for a particular cache.
- Termination records document the termination of statistics gathering for a particular cache.
- Statistics records are a blank-delimited set of numbers showing activity within the cache.

Initialization record

Initialization records are of the form:

```
mm/dd/yy hh:mm:ss id Initialization: interval n seconds
```

where

mm/dd/yy	is the month, day, and year that statistics gathering starts.
hh:mm:ss	is the hour, minute, and second that statistics gathering starts.
id	is the name of the cache the record is for
n	is the collection interval

Termination record

Termination records are of the form:

```
 mm/dd/yy hh:mm:ss id Termination
```

where

mm/dd/yy	is the month, day, and year statistics gathering stops.
hh:mm:ss	is the hour, minute, and second statistics gathering stops.
id	is the name of the cache the record is for.

Statistics record

Statistics records are of the form:

```
    mm/dd/yy hh:mm:ss id <statistics>
```

where

mm/dd/yy	is the month, day, and year the record is created.
hh:mm:ss	is the hour, minute, and second the record is created
id	is the name of the cache the record is for
<statistics>	is a blank-delimited list of statistics gathered for this cache.

The list of statistics consists of the following fields, in the following order:

Field #	Contents	Description
1	reads	Number of read operations against the cache.
2	writes	Number of write operations against the cache.
3	closes	Number of close operations on objects in the cache.
4	open read	Number of open-read operations on objects in the cache.
5	open write	Number of open-write operations on objects in the cache.
6	open write query	Number of open-write-query operations on objects in the cache.
7	read hits	Number of read hits objects in the cache.
8	write hits	Number of write hits on objects in the cache.
9	write query hits	Number of write query hits on objects in the cache.
10	initializations	Number of new sessions established with this cache.
11	terminations	Number of sessions terminated with this cache.
12	memory used	Amount of memory used by objects in the memory portion of the cache.
13	disk used	Amount of disk space used by objects in the disk portion of the cache.
14	memory available	Amount of memory still available for use by objects in the memory portion of the cache.
15	disk available	Amount of disk space still available for use by objects in the disk portion of the cache.
16	memory object count	Number of objects in the memory portion of the cache.
17	file object count	Number of objects in the disk portion of the cache.
18	session count	Number of sessions currently active against this cache.

The counters for the following items are reinitialized to zero (0) each time a record is written: reads, writes, closes, open_read, open_write, open_write_query, read_hits, write_hits, write_query_hits, initializations, terminations.

Accessing Statistics from an Application Program

An application program communicating with a cache manager can access statistics counters from a cache via the API call

- CacheGetStats.

Cache Coherency

Maintaining coherency of database and cache can be difficult. The principal problem is managing the many-to-many relationship of database tables to pages. That is, many database tables may participate in the construction of many different pages.

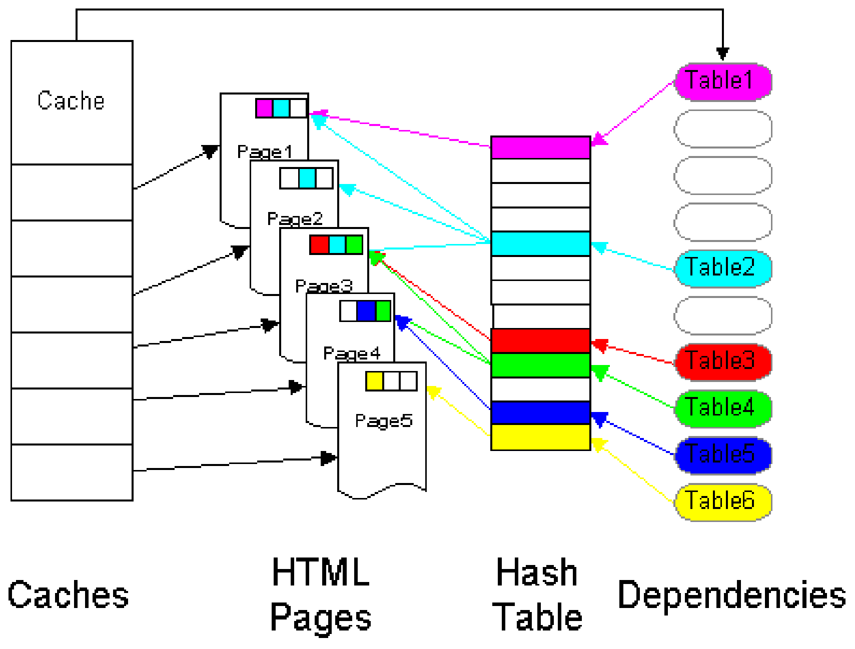
Both the program which generates the pages and the program which fetches them (httpd) must view the cache from the point of view of each page's URL. The database, however, must view the cache from the point of view of the tables and rows from which each page is constructed. The pages in cache can be said to be *dependent* on the database tables from which they are created.

To solve this problem, the cache manager maintains two different references to a page, its *primary key*, and its *dependency list*.

Each page is indexed via its primary key, assigned during a CacheOpen() call. This is the key which the httpd uses.

After a page is created, the secondary association, or *depdendency* may optionally be defined. Any number of *dependencies* may be defined. A *dependency* is an arbitrary string associated with each page. Once defined, a cache item (page) may be purged by referencing this dependency. Many pages may be associated with the same dependency, in which case, all pages referenced by that *dependency* are purged.

In the following diagram, pages 1, 2, and 3 are dependent on "table2". If "table2" which has just been updated, we can purge these three pages with a single call to CacheInvalidateRecord(... , "Table2").



The following API calls are used to manage dependencies:

- CacheAddDependency()
- CacheDeleteDependency()
- CacheInvalidateRecord()
- CacheShowDependentKeys()
- CacheFreeBstringArray()

Application Program Interface

The API is the only way to communicate with the cache manager. The cacheadm command itself is an application written the to API. This section describes the API in detail.

The API is distributed as a shared object. This permits update and modification of the cache manager, the API, and the protocols without the need for applications to be recompiled. The shared object is distributed under the name

```
cache_api.shr.o
```

API Overview

The basic idea behind the API is to view a cache as a filesystem: one opens an item for read or write, reads or writes data to it, and closes it. However, the analogy is not complete. It makes no sense to open an item for read/write; hence open is supported for READ-ONLY and WRITE_ONLY. A third mode, WRITE_ONLY_WITH_QUERY, described below, is also supported.

If an object exists in cache and is then opened in WRITE_ONLY mode, the object is deleted, its space freed, and the cache prepares itself to receive a new object in its place.

Some applications may wish to (a) query the cache to see if an item exists, and (b) if it exists, do nothing, or (c) if it does NOT exist, create a WRITE_ONLY connection so the item can be created. One way to do this is to open an item in READ_ONLY mode, and if the open indicates the object exists, close it and reopen it in WRITE_ONLY mode. To avoid the overhead of multiple opens, a special mode, WRITE_ONLY_WITH_QUERY is implemented on the open command, which does exactly this, in a single operation. If the item exists, the item is NOT opened but the token is updated to indicate its status. If it does not exist, the item is opened in WRITE_ONLY mode so the application can now create it.

To open an item, a token must be created. That token is passed to the cache manager, and, if the item exists, is returned updated with all relevant information: date of creation, date of last access, etc. This information may be ignored, or used by the application to manage cache entries directly, thus enabling applications to fully implement cache policy.

Two things must be determined before data can be accessed:

1. where is the data to be read/written within the application?
2. which cache is to be used?

Specification of which cache is to be used is done while initiating a session with the cache manager.

Specification of the location of the data is done with the token-creation calls.

There are three token-creation calls:

1. create a buffer token (data comes from a buffer),
2. create a stream token (date comes from an I/O stream), and
3. create a file token (data comes from a file).

The API implements the concept of a session, established between the application and a specific cache. An application initiates a session with a cache and performs as many operations on data within that cache as it likes (open, read, close, open, write, close, etc.) The session may be kept permanently open if desired as the cache manager is multi-threaded. This permits long-running processes (such as web servers) to maintain contact with the cache manager with a single open. An application may switch its current sessesion among multiple caches on the same physical connection with a cache switch call.

A set of API calls is implemented to manage dependencies: create, delete, purge, and query.

A set of API calls is implemented to permit administrative functions: purge an entry, flush a cache, terminate the cache manager.

Finally, a set of API calls is provided as convenience functions to manage the token, read, and write of an item within a single call.

Description

The API consists of two structures, and a set of subroutine calls. To access the definition of these, use the following #include:

```
     #include ‘‘CacheApi.h’’
```

The CacheHandle structure is an opaque pointer to a structure used by the API to communicate with the cache manager. The API never needs to access any of its fields directly.

The CacheToken structure is a c-language structure which contains data about a specific cache entry. It contains the key for the data, maintains an association between a data item and a particular cache, and contains all available data about a data in cache (age of item, etc.)

Structures

CacheHandle

CacheHandle is an opaque pointer to a structure that represents a cache.

CacheStringList

CacheStringList is an opaque pointer to a structure that represents a key. CacheHandle is an opaque pointer to a structure that represents a cache.

The CacheToken Structure

The CacheToken structure is defined thus:

```
    typedef struct _CacheToken {
        char           * key;
        int            key_len;
        int            datum_len;
        time_t         creation;
        time_t         expiration;
        time_t         last_access;
        CacheHandle    connection;
        enum CacheRc   return_code;
        enum CacheDisp disp;
    } CacheToken;
```

The CacheStatToken Structure

The CacheStatToken structure is defined thus:

```
typedef struct _CacheStatToken {
    int                         reads;
    int                         writes;
    int                         closes;
    int                         open_reads;
    int                         read_hits;
    int                         open_writes;
    int                         write_hits;
    int                          open_write_queries;
    int                          write_query_hits;
    int                          initializations;
    int                          terminations;
    int                          purges;
} CacheStatToken;
```

CacheDISP Definition

The CacheDisp constants are defined thus:

```
    enum CacheDisp{ CacheRO,      // open read only
                    CacheWO,      // open write only
                    CacheNone,    // not open
                    CacheDispMax  // max disp (bounds
                                  //    checking on xmit)
    };
```

CacheRC Definition

The CacheDisp constants are defined thus:

```
    enum CacheRc { CacheNo,          // not in cache or can’t
                                     //    be opened or buffer
                                     //    can’t be set
                   CacheExpired,     // in cache, but expired
                   CacheFound,       // found and opened as
                                     //    requested
                   CacheLocked,      // in cache, but already
                                     //     open by this process
                   CacheNA,          // communication failure
                                     //    (check errno)
                   CacheTooLong,     // buffer > max
                   CacheNotUserManaged, // user attempted to
                                     //     set buffer but the
                                     //     buffer is being
                                     //     managed by the API.
                   CacheRcMax        // max rc (bounds check
                                     //     on xmit)
    };
```

CByteString Definition

The CByteString structure is used for accessing the data returned by CacheInvalidateRecord(), CacheShowDependentKeys(), and CacheStringListNext(). These structures must be treated as READ-ONLY and never modified or freed other than by invoking CacheStringListFree().

```
     typedef struct _CByteString {
         char * data;           /* the key */
         int len;               /* length of key */
     } CByteString;
```

All fields are updated by the API only; direct manipulation by applications is not supported. The fields creation, expiration, and last_access are initialized (and there valid) only after the item has been successfully opened.

CacheAddDependency

Description:

Associate a dependency string with a cached item.

Syntax:

int CacheAddDependency(CacheHandle ch, void *key, int klen, char *dep)

where

ch	is the handle acquired from CacheInit.
key	points to string of bytes which matches the key for the item.
klen	is the length of the key
dep	is the dependency to be added

On return, the string *dep* will be associated within the cache with the item stored under *key*.

Return Codes

CacheFound	The dependency has been set.
CacheNo	The item *key* is not found in cache.
CacheNoHashing	Hashing is not enabled for this cache.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);
...
CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0;
...

CacheAddDependency(ch, ‘‘key3’’, 4, "table1");
CacheTerm(ch)
```

Transaction Log Format

Transaction log records are in the following format:

```
<date> <time> CacheAddDependency <key> <dep>
```

CacheDeleteDependency

Description:

Disassociate a dependency string from a cached item.

Syntax:

int CacheDeleteDependency(CacheHandle ch, void *key, int klen, char *dep)

where

ch	is the handle acquired from CacheInit.
key	points to string of bytes which matches the key for the item.
klen	is the length of the key
dep	is the dependency to be deleted

On return, the specified page will no longer have *dep* as a dependency.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);

CacheDeleteDependency(ch, ‘‘key3’’, 4, "table1");
CacheTerm(ch);
```

Transaction Log Format

Transaction log records are in the following format:

```
<date> <time> CacheDeleteDependency <key> <dep>
```

CacheInvalidateRecord

Description:

This call deletes all items in cache associated with the specified dependency string, returning the list of cache keys for all items deleted.

Syntax:

CacheStringList CacheInvalidateRecord(CacheHandle handle, char *dep);

where

handle	is the handle acquired from CacheInit.
dep	is the dependency string for which all items will be purged.

On return, the list of cache keys corresponding to items which were deleted in a CacheStringList. This array must be discarded later via a call to CacheStringListFree(). The individual entries in the list can be accessed via a call to CacheStringListReset() followed by a set of calls to CacheStringListNext().

Example:

```
#include ‘‘CacheAPI.h’’
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);

CacheStringList list;
list = CacheInvalidateRecord((ch, ‘‘key3’’)
...
CacheTerm(ch);
```

Transaction Log Format

For each object denoted by <key> which is invalidated, a record of the following format is generated:

```
<date> <time> CacheInvalidateRecord <dep> <key> <datum-len>
```

If no objects are invalidated as a result of the function call, a single record of the following format is generated:

```
<date> <time> CacheInvalidateRecord <dep>
```

CacheShowDependentKeys

Description:

This call returns the list of keys corresponding to cache items marked dependent on the specified string. No items are deleted.

Syntax:

CacheStringList CacheShowDependentKeys(CacheHandle ch, char *dep)

where

handle	is the handle acquired from CacheInit.
dep	is the dependency string for which keys will be returned.

On return, the list of cache keys corresponding to items which were deleted in a CacheStringList. This array must be discarded later via a call to CacheStringListFree(). The individual entries in the list can be accessed via a call to CacheStringListReset() followed by a set of calls to CacheStringListNext().

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);

CacheStringList list;
list = CacheShowDependentKeys((ch, ‘‘key3’’) // fetch keys
int count = list.reset();                     // reset list and get count
for (int i = 0; i < count; i++) {
   CByteString *cbs = list.next();            // fetch next item
}
CacheStringListFree(list);                    // free all storage
...
CacheTerm(ch);
```

Transaction Log Format

Transaction log records are in the following format:

```
<date> <time> CacheQueryDependentKeys <dep>
```

CacheStringListFree

Description:

This call frees storage acquired via CacheInvalidateRecord() or CacheShowDependentKeys(). This storage is managed by the API and ***must*** be returned with this call, not free() or delete.

Syntax:

CacheStringListFree(CacheStringList);

where

array	is a CacheStringList acquired from one of the other cache API calls.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);

CacheStringList list;
list = CacheShowDependentKeys((ch, ‘‘key3’’)
...
CacheStringListFree(list);
CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheStringListReset

Description:

This call initializes an interator inside the CacheStringList() structure, returning the number of items in the structure.

Syntax:

int CacheStringListReset(CacheStringList);

where

array is a CacheStringList acquired from one of the other cache API calls.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);

CacheStringList list;
list = CacheShowDependentKeys((ch, ‘‘key3’’)
    int count = CacheStringListReset(list);
...
CacheStringListFree(list);
CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheStringListNext

Description:

This returns the next CByteString from a CacheStringList() structure and advances the internal pointer to the next item.

Syntax:

CByteString *CacheStringListNext(CacheStringList);

where

array is a CacheStringList acquired from one of the other cache API calls.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);

CacheStringList list;
list = CacheShowDependentKeys((ch, ‘‘key3’’)
int count = CacheStringListReset(list);
for (int i = 0; i < count; i++) {
   CByteString *cbs = CacheStringListNext(list);
   printf("Key=%s\b", cbs->key);
}
...
CacheStringListFree(list);
CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheInit

Description:

CacheInit initializes a session between the application and a specific cache. It returns a CacheHandle, required for all subsequent communication with the cache manager.

Syntax:

CacheHandle CacheInit(char *machine, int port, char *cache, int timeout=30);

where

machine	is the name of the machine running the cache manager.
port	is the TCP/IP port the cache manager is defined to use.
cache	is the name of one of the configured caches, and
timeout	is the maximum length of time in seconds that a read or write should be left pending with no response. If this time is exceeded, the connection is closed.

On return, the *CacheHandle* is either 0 or non-zero; if 0, the open failed, and if non-zero, represents a session between the API and the cache manager. If an error occurred, errno is set to determine the cause of the problem. The following errno values are set:

ENOENT	The specified cache was not found. The cache parameter is invalid.
ENOTREADY	The specified cache was found, but is marked inactive.

Any other errno is the errno from any possible failed socket or connect system calls.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey’’,
                           7175,
                           ‘‘cache0’’,
                            30);
...
```

Transaction Log Format

Transaction log records are in the following format:

```
<date> <time> CacheInit <status>
```

where <status> is 1 if the cache is active and 0 otherwise.

CacheTerm

Description:

CacheTerm call is used to terminate a session. It must be passed an open CacheHandle. On return, the CacheHandle is no longer vali/td>

Syntax:

void CacheTerm(CacheHandle ch)

where

ch is the handle acquired from CacheInit.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheTerm(‘‘gallifrey,
                           7175,
                           ‘‘cache0’’,
                            30);
...
CacheTerm(ch);
```

Transaction Log Format

Transaction log records are in the following format:

```
<date> <time> CacheTerm
```

CacheSwitch

Description:

The CacheSwitch call is used to switch sessions between caches on the same physical connection. It eliminates the need for CacheTerm and another CacheInit if an application wishes to manage multiple caches on the same connection.

Syntax:

CacheHandle CacheSwitch(CacheHandle ch, char *id;

where

ch	is a valid, open connection, acquired on a previous CacheInit.
id	is the name of a valid cache.

On return, the CacheHandle is either zero or non-zero and may be used in further cache operations. In either case, the previous connection is closed. If an error occurred, errno is set to determine the cause of the problem. The following errno values are set:

ENOENT	The specified cache was not found. The cache parameter is invalid.
ENOTREADY	The specified cache was found, but is marked inactive.

Any other errno is the errno from any possible failed socket system calls.

Example:

```
#include ‘‘CacheAPI.h’’
  ...
  CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);
  ...
  CacheHandle ch1 = CacheSwitch(ch, ‘‘cache1’’);
  ...
  CacheTerm(ch1);
```

Transaction Log Format

A transaction record of the following format is placed in the transaction log for the old cache:

```
<date> <time> CacheTerm
```

A transaction record of the following format is placed in the transaction log for the new cache:

```
<date> <time> CacheInit
```

CacheBufferToken

Description:

The CacheBufferToken call defines a token used to read or write data from an application buffer. It is used to specify the following:

- Data to be used as a key for the datum and the length of the key,
- The location of a buffer to be used for I/O (the application will move data into and out of this buffer as appropriate) and the length of the buffer
- The maximum size of the data item.

Syntax:

CacheToken *CacheBufferToken(void *key, int klen, char *buffer, int buffer_len, int data_len);

where

key	is a pointer to a string of bytes to be used as the key for this data. This is not a c-language string and may consist of any sequence of bytes, including imbedded ‘\0’.
klen	is the length in bytes of the key,
buffer	is a pointer to the buffer to be used for moving data,
buffer_len	is the length of the buffer,
data_len	is the maximum size of the data. This is the maximum number of bytes the API will move for this item. If the data consists of fewer bytes, the cache manager will adjust the data item appropriately as if data_len had been specified exactly. If the data is to be read from cache, this may be specified as 0 (i.e. unknown).

On return a pointer to a CacheToken is returned. This is to be used in a subsequent CacheOpen() call.

Example:

This example creates a CacheToken for use with a buffer of length 4096 and a data item no larger than 20,000 bytes, to be cached under the key ‘‘key0’’.

```
   #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                7175,
                                ‘‘cache0’’,
                                 30);
     ...
     char buf[4096];
     CacheToken *ct = CacheBufferToken(‘‘key0’’, 4, buf, sizeof(buf), 20000);
     ...
     CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheSetBuffer

Description:

The CacheSetBuffer command allows the application to specify a different buffer than that set by a previous CacheBufferToken call. This allows a data item to be written from an array of strings.

Syntax:

int CacheSetBuffer(CacheToken *tok, char *buffer, int len);

where

tok	is the CacheToken created in a previous CacheBufferToken call.
buffer	is a pointer to the new buffer.
len	is the length of the new buffer.

Example:

This example assumes a routine called make_strings returns an array of strings, the first two elements of which are to be written to cache under key0;

```
  #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                7175,
                                ‘‘cache0’’,
                                 30);
     ...
     char **strings = make_strings();
     CacheToken *ct = CacheBufferToken(‘‘key0’’,
                                       4,
                                       strings[0],
                                       strlen(strings[0],
                                       20000);
     CacheWrite...
     CacheSetBuffer(ct, strings[1], strlen(strings[1]);
     CacheWrite...
     ...
     CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheGetBuffer(CacheToken *tok)

Description:

This call returns a pointer to the buffer owned by the specified token.

Syntax:

char *CacheGetBuffer(CacheToken *tok);

where

tok is a pointer to a valid token

A pointer to the buffer is returned.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);
...
char buf[4096];
CacheToken *ct = CacheBuffer(‘‘key0’’, 4, buf, sizeof(buf), 20000);
...
char *buf1 = CacheGetBuffer(ct);
...
CacheTerm(ch)
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheFileToken

Description:

This call creates a CacheToken for use in reading a file into cache, or writing an item out of cache into a file.

Syntax:

CacheToken *CacheFileToken(void *key, int klen, char *filename);

where

key	is a pointer to a string of bytes to be used as a key for the item,
klen	is the length in bytes of the key, and
filename	is the name of the file to be read/written. Note that this is a file name, not any sort of file handle. The API will manage all file operations including open and close.

On return a pointer to a CacheToken is returned. This is to be used in a subsequent CacheOpen() call.

Example:

This example prepares a file ‘‘foo’’ for use by the cache API.

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
CacheToken *ct = CacheFileToken(‘‘key1’’, 3, ‘‘foo’’);
...
CacheTerm(ch)
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheStreamToken

Description:

This call creates a CacheToken for use in reading from an open file descriptor into cache, or writing from cache to an open file descriptor.

Syntax:

CacheToken *CacheStreamToken(void *key, int klen, int stream, int dlen);

where

key	is a pointer to a byte string to be used as the key for this item,
klen	is the length of the key in bytes
stream	is an open file descriptor, and
dlen	is the maximum number of bytes to be read or written. If the item is to be read from cache this may be specified as 0 (i.e., unknown).

On return a pointer to a CacheToken is returned. This is to be used in a subsequent CacheOpen() call.

Example:

This example prepares a token for writing from cache to stdout.

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                           7175,
                           ‘‘cache0’’,
                            30);
...
CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0);
...
CacheTerm(ch)
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheFreeToken

Description:

This call frees a token and any associated resources which were acquired by the API. Note that the user is responsible for closing streams (from CacheStreamToken) or freeing buffers (from CacheBufferToken) acquired by the application.

Syntax:

void CacheFreeToken(CacheToken *tok);

where

tok is a token acquired from one of the token management calls.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0;
...
CacheFreeToken(ct);
CacheTerm(ch)
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheOpen

Description:

This call associates a session (from CacheInit) with a token (from CacheOpen) and establishes the direction of data flow (from cache to API or from API to cache).

Syntax:

CacheRc CacheOpen(CacheToken *token, CacheHandle handl, CacheDisp disp);

where

token	is a token acquired from one of the Cache...Token calls.
handle	is a handle acquired from CacheInit or CacheSwitch.
disp	is one of CacheRO, CacheWO.

The semantics of disp are:

CacheRO	open the item in read-only mode. If the item exists, the token is updated with the actual length of the data item in cache, the time the item was created (or last written), the time the item expires, and the time the data was last accessed
CacheWO	open the item in write-only mode. If the item already exists it is deleted and a new entry is created to hold the new data.

The following codes can be returned by this call:

CacheFound the item is found in the cache if disp is CacheRO. The item is successfully opened

	if disp is CacheWO.
CacheNo	the item is not in cache, or can’t be opened, or a buffer for data transfer cannot be established.
CacheExpired	The item is found in cache but is marked expired. The item is successfully opened.
CacheLocked	The item is found in cache but is already opened by this process. The item is not reopened by this call.
CacheNA	The cache is not available. Probably a communication failure.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0;
CacheRc rc = CacheOpen(ct, ch, CacheRO);
...
CacheFreeToken(ct);
CacheTerm(ch)
```

Transaction Log Format

If the item is already open, a transaction record in the following format is generated:

```
<date> <time> CacheOpen_already_open <key>
```

If disp=CacheRO and the item is found, a transaction record in the following format is generated:

```
<date> <time> CacheOpen_RO <key> <datum-len>
```

If disp=CacheRO and the item is not found, a transaction record in the following format is generated:

```
<date> <time> CacheOpen_RO <key>
```

If disp=CacheWO, a transaction record in the following format is generated:

```
<date> <time> CacheOpen_RO <key> <datum-len>
```

CacheClose

Description:

This call dissociates the connection between a cache session and a token.

Syntax:

void CacheClose(CacheToken *ct);

where

ct is a token which has been opened by CacheOpen.

Example:

```
#include ‘‘CacheAPI.h’’
 ...
 CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
 ...
 CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0;
 CacheRc rc = CacheOpen(ct, ch, CacheRO);
 ...
 CacheClose(ct);
 CacheFreeToken(ct);
 CacheTerm(ch)
```

Transaction Log Format

A transaction record in the following format is generated:

```
<date> <time> CacheClose <key> <datum-len>
```

If the cache API is managing buffer storage for the token and the disp field of the token is CacheWO, a CacheFlush API call is made before the CacheClose. A transaction record of the following format is generated before the CacheClose transaction record:

```
<date> <time> CacheWrite <key> <datum-len>
```

CacheRead

Description:

This call causes data to be transferred from the cache to the location specified by the token.

Syntax:

int CacheRead(CacheToken *token);

where

token is a token opened in CacheRO mode.

The actual number of bytes transferred is returned. If the call fails, the return code is -1.

Example:

s reads the data in cache associated with key key3 and writes the contents to stdout. Note that except for error checking, the last five lines of this example are now a complete program.

```
#include ‘‘CacheAPI.h’’
 ...
 CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
 ...
 CacheToken *ct = CacheStreamToken(‘‘key3’’, 4, 1, 0;
 CacheRc rc = CacheOpen(ct, ch, CacheRO);
 int rc1 = CacheRead(ct);
 CacheClose(ct);
 CacheFreeToken(ct);
 CacheTerm(ch);
```

Transaction Log Format

A ransaction record in the following format is generated:

```
<date> <time> CacheRead <key> <bytes-read>
```

CacheWrite

Description:

This call transfers data from the location specified by the token into the cache.

Syntax:

int CacheWrite(CacheToken *token);

where

token is a token opened in CacheWO mode.

The actual number of bytes transferred is returned. If the call fails, the return code is -1.

Example:

This writes 50 bytes from stdin to an item in cache associated with key key4.

```
#include ‘‘CacheAPI.h’’
 ...
 CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
 ...
 CacheToken *ct = CacheStreamToken(‘‘key4’’, 4, 0, 50)
 CacheRc rc = CacheOpen(ct, ch, CacheWO);
 int rc1 = CacheWrite(ct);
 CacheClose(ct);
 CacheFreeToken(ct);
 CacheTerm(ch);
```

Transaction Log Format

A ransaction record in the following format is generated:

`<date> <time> CacheWrite <key> <bytes-written>`

CacheBuffer

Description:

This is a convenience call, which prepares a buffer for reading or writing from cache, freeing the user from the task of managing buffers and coding CacheBufferToken and CacheOpen calls. It is used in conjunction with CacheAppend to actually transfer data/td>

Syntax:

CacheToken*CacheBuffer(CacheHandle ch, char *key, int keylen, int datalen, CacheDisp disp, CacheRc *rc);

where

ch	is a handle acquired from CacheInit or CacheSwitch,
key	is a byte string to be used as a key,
keylen	is the length in bytes of the key,
datalen	is the length of buffer to use,
disp	is CacheRO or CacheWO, and
rc	is the return code from the implicit CacheBufferToken call done by this routine.

This routine returns either 0 (failure) or a CacheToken to be used on subsequent CacheAppend calls.

Example:

This example prepares to read from the cache entry key5.

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
CacheRc rc = CacheNo;
CacheToken *ct = CacheBuffer(CacheHandle ch,
                    ‘‘key5’’, 4, 0, CacheRO, &rc);
...
CacheClose(ct);
CacheFreeToken(ct);
CacheTerm(ch);
```

Transaction Log Format

This function generates identical log records as the corresponding CacheOpen call would generate.

CacheAppend

Description:

This is a convenience function for use in conjunction with CacheBuffer to ease data transfer into and out of API buffers.

Syntax:

int CacheAppend(CacheToken *ct, void *buffer, int len);

where

ct	is the token returned from a previous CacheBuffer call,
buffer	is the location data is to be moved into, in the case of CacheRO, or from, in the case of CacheWO,
len	is the number of bytes to be moved (which may be different from the actual size of the data item).

Example:

This example reads 800 bytes in 80 byte increments from key5 in cache and writes them to stdout.

```
#include ‘‘CacheAPI.h’’
 ...
 CacheHandle ch = CacheInit(‘‘gallifrey,
                             7175,
                             ‘‘cache0’’,
                              30);
 ...
```

```
     char buf[80];
     CacheRc rc = CacheNo;
     CacheToken *ct = CacheBuffer(CacheHandle ch,
        ‘‘key5’’, 4, 1, CacheRO, &rc);
     for (int i = 0; i < 10; i++) {
        CacheAppend(ct, buf, 80);
        fwrite(stdout, ‘‘%s’’, buf);
     }
     CacheClose(ct);
     CacheFreeToken(ct);
     CacheTerm(ch);
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

CacheFile

Description:

This is a convenience function to read an entire item from cache to disk, or write an entire file from disk to cache in a single call.

Syntax:

int CacheFile(CacheHandle ch, void *key, int klen, char *filename, CacheDisp disp, CacheRc *rc);

where

ch	is a handle acquired from CacheInit or CacheSwitch,
key	is a byte string comprising the key,
klen	is the length in bytes of the key,
filename	is the name of the file to be read or written,
disp	is either CacheRO or CacheWO, and
rc	is the return code returned from the implicit CacheRead or CacheWrite.

The number of bytes of data actually transferred is returned, or -1 if the call fails.

Example:

This example copies the data in cache under key6 into the file called data6.

```
  #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                 7175,
                                 ‘‘cache0’’,
                                  30);
     ...
     CacheRc rc = CacheNo;
     int rc1 = CacheFile(ch, ‘‘key6’’, 4, ‘‘data6’’,
                 CacheRO, &rc);

     CacheTerm(ch);
```

Transaction Log Format

This function generates identical log records as the corresponding CacheOpen and CacheRead or CacheWrite calls would generate.

CacheStream

Description:

This is a convenience function to permit the reading and writing data between cache and an open file descriptor in a single call.

Syntax:

int CacheStream(CacheHandle ch, void *key, int klen, int dlen, int stream, CacheDisp disp, CacheRc *rc);

where

ch	is a handle acquired in CacheInit or CacheSwitch,
key	is a byte string representing the key,
klen	is the number of bytes in the key,
dlen	is the number of bytes to be transferred (or 0, if disp = CacheRO),
stream	is an open file descriptor,
disp	is either CacheRO or CacheWO,
rc	is the return code from the implicit CacheRead or CacheWrite.

The number of bytes of data actually transferred is returned, or -1 if the call fails.

Example:

This example reads the contents of the data associated with key7 and writes it to stdout.

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                                ‘‘cache0’’,
                                 30);
...
CacheRc rc = CacheNo;
int rc1 = CacheStream(ch, ‘‘key7’’, 4, 0, 1, CacheRO, &rc);

CacheTerm(ch);
```

Transaction Log Format

This function generates identical log records as the corresponding CacheOpen and CacheRead or CacheWrite calls would generate.

CachePurge

Description:

This call causes an entry to be purged from cache.

Syntax:

int CachePurge(CacheToken *token);

where

token is an open token. Either CacheRO or CacheWO may be used to open the token. The CacheOpen is required to insure the item exists and to associate a session with the item.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
CacheToken *ct = CacheBufferToken(‘‘key0’’, 4,  0, 0, 0);
CacheOpen(ch, ct, CacheRO);
... // determine if it should be purged by
    // looking at the returned token
CachePurge(ct);
CacheTerm(ch);
```

A transaction record in the following format is generated:

```
<date> <time> CachePurge <key> <datum-len>
```

CacheFlush

Description:

This call forces a physical write of any buffered data written with any of the calls which cause data to be transferred to the cache.

Although closing a token will force the buffers to be flushed, you should always do an explicit flush so you can examine the return code and insure all data was safely transferred over the network./td>

Syntax:

int CacheFlush(CacheToken *ct);

where

token is a token open in CacheWO mode.

Example:

```
#include ‘‘CacheAPI.h’’
...
CacheHandle ch = CacheInit(‘‘gallifrey,
                            7175,
                            ‘‘cache0’’,
                             30);
...
char buf[80];
CacheRc rc = CacheNo;
CacheToken *ct = CacheBuffer(CacheHandle ch,
   ‘‘key5’’, 4, 0, CacheWO, &rc);
for (int i = 0; i < 10; i++) {
   CacheAppend(ct, buf, 80);
   fwrite(stdout, ‘‘%s’’, buf);
}
CacheFlush(ct); // insure all data safely written
CacheClose(ct);
CacheFreeToken(ct);
CacheTerm(ch);
```

A transaction record in the following format is generated:

```
<date> <time> CacheWrite <key> <datum-len>
```

CacheClear

Description:

Cause all data items in a specific cache to be deleted unconditionally.

Syntax:

int CacheClear(CacheHandle ch);

where

ch is a handle to a cache opened by CacheIint or CacheSwitch.

Example:

Purge all items in cache ‘‘cache0’’.

```
 #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                 7175,
                                 ‘‘cache0’’,
                                  30);
     CacheClear(ch);
     CacheTerm(ch);
```

A transaction record in the following format is generated:

```
<date> <time> CacheClear
```

hr>

CacheGetStats

Description:

This call returns a pointer to a structure containing statistics from a cache.

Syntax:

CacheStatToken * CacheGetStats(CacheHandle ch, int reset_counters);

where

ch	is a handle acquired from CacheInit or CacheSwitch.
reset_counters	is 1 if the cache statistics counters should be reset after they are sent to the API program, 0 otherwise.

Example:

```
     #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                7175,
                                ‘‘cache0’’,
                                 30);
     ...
     CacheStatToken *token = CacheGetStats(ch, 0);
     ...
     CacheFreeStatToken(ct);
     ...
     CacheTerm(ch)
```

A transaction record in the following format is generated:

```
<date> <time> CacheGetStats
```

CacheFreeStatToken

Description:

This call frees a statistics token returned by CacheGetStats.

Syntax:

void CacheFreeStatToken(CacheStatToken * tok);

where

tok is a structure returned by CacheGetStats.

Example:

```
     #include ‘‘CacheAPI.h’’
     ...
     CacheHandle ch = CacheInit(‘‘gallifrey,
                                7175,
                                ‘‘cache0’’,
                                 30);
     ...
     CacheStatToken *token = CacheGetStats(ch, 0);
     ...
     CacheFreeStatToken(ct);
     ...
     CacheTerm(ch)
```

Transaction Log Format

This API function does not communicate with the cache manager daemon and hence does not affect the transaction log.

Sample configuration file

For reference, the following config file was used for verifying the test program described above.

```
cache-manager {
    port = 7175
    connection-timeout = 0
    logging = yes
    log = /u/challngr/cachemgr/cachemgr.log
    wrap-log = yes
    log-size = 64000
    trace-flags = D_ALL
}

test1
{
    root = /u/challngr/cachemgr/cache0
    caching = on
    mem-size = 10MB
    fs-size = 5MB
    datum-memory-limit = 5KB
    datum-disk-limit = 500KB
    lifetime = 6000000
    check-expiration = 150
    stat-file = /u/challngr/cachemgr/cache0.stats
    stat-interval = 60
}

test2: test1
{
    root = /u/challngr/cachemgr/cache1
    stat-file = /u/challngr/cachemgr/cache1.stats
}
```