RPC Cache

This document gives a brief introduction to the caching mechanisms in the sunrpc layer that is used, in particular, for NFS authentication.

Caches

The caching replaces the old exports table and allows for a wide variety of values to be caches.

There are a number of caches that are similar in structure though quite possibly very different in content and use. There is a corpus of common code for managing these caches.

Examples of caches that are likely to be needed are:

- mapping from IP address to client name
- mapping from client name and filesystem to export options
- mapping from UID to list of GIDs, to work around NFS's limitation of 16 gids.
- mappings between local UID/GID and remote UID/GID for sites that do not have uniform uid assignment
- mapping from network identify to public key for crypto authentication.

The common code handles such things as:

- general cache lookup with correct locking
- supporting 'NEGATIVE' as well as positive entries
- allowing an EXPIRED time on cache items, and removing items after they expire, and are no longer in-use.
- making requests to user-space to fill in cache entries
- allowing user-space to directly set entries in the cache
- delaying RPC requests that depend on as-yet incomplete cache entries, and replaying those requests when the cache entry is complete.
- clean out old entries as they expire.

Creating a Cache

- A cache needs a datum to store. This is in the form of a structure definition that must contain a struct cache_head as an element, usually the first. It will also contain a key and some content. Each cache element is reference counted and contains expiry and update times for use in cache management.
- A cache needs a "cache_detail" structure that describes the cache. This stores the hash table, some parameters for cache management, and some operations detailing how to work with particular cache items.

  The operations are:

  struct cache_head *alloc(void)
  : This simply allocates appropriate memory and returns a pointer to the cache_detail embedded within the structure

  void cache_put(struct kref *)
  : This is called when the last reference to an item is dropped. The pointer passed is to the 'ref' field in the cache_head. cache_put should release any references create by 'cache_init' and, if CACHE_VALID is set, any references created by cache_update. It should then release the memory allocated by 'alloc'.

  int match(struct cache_head *orig, struct cache_head *new)
  : test if the keys in the two structures match. Return 1 if they do, 0 if they don't.

  void init(struct cache_head *orig, struct cache_head *new)
  : Set the 'key' fields in 'new' from 'orig'. This may include taking references to shared objects.

  void update(struct cache_head *orig, struct cache_head *new)
  : Set the 'content' fileds in 'new' from 'orig'.

  int cache_show(struct seq_file *m, struct cache_detail *cd, struct cache_head *h)
  : Optional. Used to provide a /proc file that lists the contents of a cache. This should show one item, usually on just one line.

  int cache_request(struct cache_detail *cd, struct cache_head *h, char **bpp, int *blen)
  : Format a request to be send to user-space for an item to be instantiated. *bpp is a buffer of size *blen. bpp should be moved forward over the encoded message, and *blen should be reduced to show how much free space remains. Return 0 on success or <0 if not enough room or other problem.

  int cache_parse(struct cache_detail *cd, char *buf, int len)
  : A message from user space has arrived to fill out a cache entry. It is in 'buf' of length 'len'. cache_parse should parse this, find the item in the cache with sunrpc_cache_lookup_rcu, and update the item with sunrpc_cache_update.

- A cache needs to be registered using cache_register(). This includes it on a list of caches that will be regularly cleaned to discard old data.

Using a cache

To find a value in a cache, call sunrpc_cache_lookup_rcu passing a pointer to the cache_head in a sample item with the 'key' fields filled in. This will be passed to ->match to identify the target entry. If no entry is found, a new entry will be create, added to the cache, and marked as not containing valid data.

The item returned is typically passed to cache_check which will check if the data is valid, and may initiate an up-call to get fresh data. cache_check will return -ENOENT in the entry is negative or if an up call is needed but not possible, -EAGAIN if an upcall is pending, or 0 if the data is valid;

cache_check can be passed a "struct cache_req*". This structure is typically embedded in the actual request and can be used to create a deferred copy of the request (struct cache_deferred_req). This is done when the found cache item is not uptodate, but the is reason to believe that userspace might provide information soon. When the cache item does become valid, the deferred copy of the request will be revisited (->revisit). It is expected that this method will reschedule the request for processing.

The value returned by sunrpc_cache_lookup_rcu can also be passed to sunrpc_cache_update to set the content for the item. A second item is passed which should hold the content. If the item found by _lookup has valid data, then it is discarded and a new item is created. This saves any user of an item from worrying about content changing while it is being inspected. If the item found by _lookup does not contain valid data, then the content is copied across and CACHE_VALID is set.

Populating a cache

Each cache has a name, and when the cache is registered, a directory with that name is created in /proc/net/rpc

This directory contains a file called 'channel' which is a channel for communicating between kernel and user for populating the cache. This directory may later contain other files of interacting with the cache.

The 'channel' works a bit like a datagram socket. Each 'write' is passed as a whole to the cache for parsing and interpretation. Each cache can treat the write requests differently, but it is expected that a message written will contain:

- a key
- an expiry time
- a content.

with the intention that an item in the cache with the give key should be create or updated to have the given content, and the expiry time should be set on that item.

Reading from a channel is a bit more interesting. When a cache lookup fails, or when it succeeds but finds an entry that may soon expire, a request is lodged for that cache item to be updated by user-space. These requests appear in the channel file.

Successive reads will return successive requests. If there are no more requests to return, read will return EOF, but a select or poll for read will block waiting for another request to be added.

Thus a user-space helper is likely to:

```
open the channel.
  select for readable
  read a request
  write a response
loop.
```

If it dies and needs to be restarted, any requests that have not been answered will still appear in the file and will be read by the new instance of the helper.

Each cache should define a "cache_parse" method which takes a message written from user-space and processes it. It should return an error (which propagates back to the write syscall) or 0.

Each cache should also define a "cache_request" method which takes a cache item and encodes a request into the buffer provided.

> **Note**
>
> If a cache has no active readers on the channel, and has had not active readers for more than 60 seconds, further requests will not be added to the channel but instead all lookups that do not find a valid entry will fail. This is partly for backward compatibility: The previous nfs exports table was deemed to be authoritative and a failed lookup meant a definite 'no'.

request/response format

While each cache is free to use its own format for requests and responses over channel, the following is recommended as appropriate and support routines are available to help: Each request or response record should be printable ASCII with precisely one newline character which should be at the end. Fields within the record should be separated by spaces, normally one. If spaces, newlines, or

nul characters are needed in a field they much be quoted. two mechanisms are available:

- If a field begins 'x' then it must contain an even number of hex digits, and pairs of these digits provide the bytes in the field.
- otherwise a in the field must be followed by 3 octal digits which give the code for a byte. Other characters are treated as them selves. At the very least, space, newline, nul, and " must be quoted in this way.