Design Document of "Maintaining file consistency in GNUtella Style Peer to Peer File Sharing System"

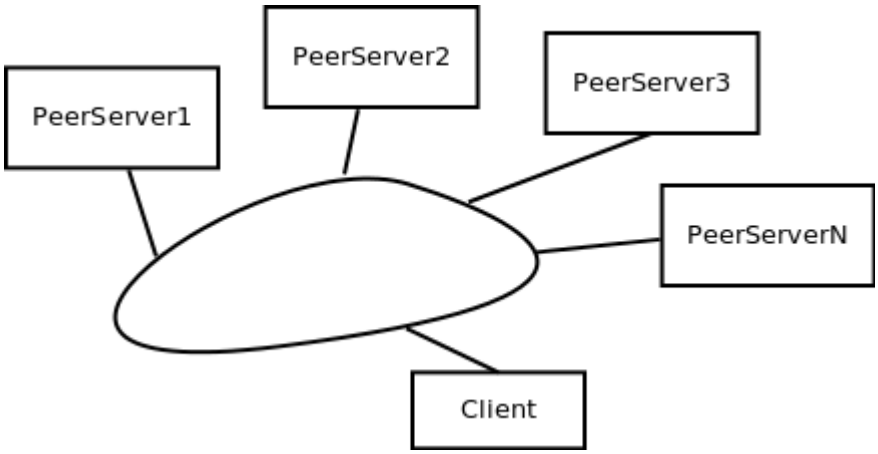
Abstract

This document describes the design and additional changes made to the components of GNUtella Style Peer-to-Peer File Sharing System. The main components of the system are :

1. Peer Server : This is the peer server which is running on every host which is part of the filesharing system. It is started with a directory that it would like to share and the list of peers that it knows.
2. Peer Client : This is the client application which will contact one of the peer servers (typically the one on the localhost) to find a list of the peers which are serving the request file and then contacts one of the peers listed by the index server to fetch the file.

The main changes done are as below :

1. Added methods validate() for pull method, invalidate() for push method
2. Modified the records passed around to include TTR, mtime, primary-flag
3. validate_thread() to validate cached files in case of pull
4. trigger_thread() to simulate updates at random intervals and on random files.
5. Added command ***update*** to request update of a particular



Design of the Peer Server :

Related Source files :
obtain.x – The RPC description of the services
obtain_xdr.c – XDR routines generated by rpcgen
obtain_clnt.c – Client stubs generated by rpcgen
obtain_client.c – Sample client code (handwritten to test indexing)
obtain_server.c – Server logic which implements the actual service
obtain_svc_mt.c – The Multithreaded code which handles multiple requests.
Makefile.obtain – The Makefile to compile the below listed executables.

Related Executables :
obtain_server – The Peer server which maintains the registry of files registered (allows querying) and serves the files.
obtain_client – The client app which the user can use to search for a file and fetch it.

The Peer server is responsible for the following :
- maintain a local registry of all the files that it is serving and all the cached results from previous queries.
- when a client queries for a file it relays the request to it is peers and also searches it's local cache.
- respond to queries from peers about the files that it is serving or knows about who else is serving it. (It uses the local cache for this purpose)

For the above mentioned servcies we have implemented the following RPC services : - search() , b_query(), b_hitquery()

Modifications to file record in the index file :

We updated the index record to capture mtime, version number, primary-flag. The record now is as below:

<revnumber> <primaryflag> <TTR> <mtime> <hostname>

Also, all the corresponding methods (search, b_query, b_hitquery) have been modified to propagate these values.

Push Method :

For this method of consistency, we use the version number recorded in the index record. For every update call received we increment the version and propagate the changes to the peers using invalidate() RPC call.
On the peers which receive this RPC call, they update their local records with the new version number.

Ensuring that only valid records are sent in Push method :

As part of sending the response to clients we search for the origin server record in the local index file. We then make sure that only records with version number equal to that of the origin server is returned. The routine corresponding to this is valid_rec().

Pull Method :

This method is slightly more cumbersome. We create a validate_thread() which does the following :

- For every file in CACHE_DIR :
  - It finds the record of the origin server for the file : calls find_origin_rec()
  - Call stat() for the CACHE_DIR/file.
  - Checks if (current_time - mtime) > TTR
    - Call validate(filename, vernumber)
    - If the newversion is different :
      - update_rec().
  - sleep(5)
    - Ideally this should depend on the next refresh.

Ensuring that only valid records are sent in Pull Method :

In this method to ensure that only the valid records are returned we do the following :

- Find the origin server record.
- Only return the record if the mtime of the record is later compared to that of origin server.
- Also, if the mtime of the cached file is older than TTR secs (compared to current time) it is potentially outdated. Hence we don't add such a record.

valid_rec() : The routine that makes sure the record is valid :

This is the routine which is responsible for ensuring that a given file record is valid. This is the common routine which is used for both push and pull method. Based on the modes enabled (push and or pull) it check compares the version number and the modification time and the TTR as described in the above sections.

Method to Trigger updates automatically :

Added the option “-t <delay>”. When this option is specified we spawn a new thread and it does the following :

- sleep for a random time (between 0 and delay secs)
- Randomly pick a file and update it.
  - For this we need a count of files we are serving : filecount
  - generate a random number : fileno = between 0 and filecount
  - open the sharedir, call readdir() fileno times. This will get us the random file that we picked.
  - Call update_1_svc() on it.

Adding Files to local cache : addcache_1_svc() :

We modified to the code to maintain a local cache directory “/tmp/cachedir” which will hold the cached files. The client code (obtain_client.c) is modified to add the downloaded file to the local cache. For this we added a new RPC method addcache(). The obtain_client on successfully downloading the file calls the addcache() call to

the local server.

The server on receiving the call does the following :

- Checks if it is the primary server for the file being added to the local cache.
  If yes, there is no need for it to cache it. Hence, returns.
- Else, copies the given file to “/tmp/cachedir”. And adds a record to index file (by calling add_peer()).
  NOTE : We copy the file into the cachedir by default as assume that it would be the latest copy.

Refreshing the local cache :

Since, the obtain_client by default always adds the downloaded file to the local cache we did not add a separate option to refresh the local cached files. However, it is a trivial change to the obtain_client to make this optional and allow user to update the local cache on demand.

Command to update a file on demand : update.c

We added a new command ***update*** which take a filename to be updated. This calls the update() RPC call on the local server and the local server updates the revision of the file and broadcasts the new version.

RPC Call - search – outline of the logic

This RPC call allows the peer download application to query the peer server for a particular file. The following are the arguments that are passed by the download app :

```
struct query_req {
  string fname<MAXNAME>; /* Name of the file that the app is looking for */
  int count;  /* Max number of peers names that return value can contain */
 };
```

The peer server does the following :

- It first builds a query request (b_query_req) with a fixed TTL of 5 hops.
- It propagates this query to its peers (b_query_propagate() calls the b_query_1() for this). Here we have an optimization.
  If the peers name is already in the local cache for the requested file we donot relay the query to it.
  The reasoning is as below :
  - If the peers name is already in the cache for the requested file, it means in the past we had already queried that peer for that file and it had relayed it to its neighbours and all the results have been sent back to this node and have been cached.
    As part of relaying the query we also add this request (with a unique sequence number) to the local linked list of pending queries (pending – is the name of the linked list).
- After relaying the query we wait for the results to arrive. Here we do a timed-wait (timeout being 5 secs) on a condition variable. This condition variable will be signalled when all the peers whom we sent the request respond back. However, there are cases when the peers don't respond back (because, they or their peers don't have that file). In such case the wait times out and sends whatever names are present in the local cache. The localcache is populated by b_hitquery() which is the response from the peers. This will be explained in the future sections.
- Once the thread exits from timed wait on the condition variable it opens the local cache and builds a list of peers serving that file. For this it tries to open the file "/tmp/indsvr/<fname>". If found it reads the contents (which is a list of peers that are serving that file) in the following structure :

```
struct query_rec {
    char        fname[MAXNAME];
    int         count;
    char        peers[BUFSIZE];
    int         pflags[MAXCOUNT];
    int         vers[MAXCOUNT];
    my_time_t   ttrs[MAXCOUNT];
    my_time_t   mtimes[MAXCOUNT];
    long        off;
    int         eof;
};
```

Notice, the additional fields for ttr, mtimes and vers. Here the valid_rec() call is made for every record to ensure that it still valid based on push or pull mechanisms (as described above for valid_rec()).

NOTE : We faced a problem with the XDR routines where the calls kept failing with "Can't encode arguments". On debugging we found that it failed to encapsulate the arguments when MAXCOUNT was 16. After some experiments we found that it worked only with MAXCOUNT = 1. This meant that we had to change the search_1_svc() and obtain_client.c code to handle multiple calls to keep getting next entries. We added the offset and eof fields to handle this part. This also meant that the number records that could be transmitted per call reduced from 16 to 1. And hence, the number of b_hitquery() calls increased drastically and destabilized the server code.

propagate_invalidate – Helper routine to relay invalidate messages :

This is the helper routine used by update and invalidate RPC calls to relay the query to all the peers if the server has not already processed this query. It uses a linked list (ipending) of all the current requests that have been seen by the server in the last one minute (reaper_thread() explained later prunes the entries older than one minute).

Below is the outline of the logic :

- Checks if we already have received this query (check on the pending list for the msg_id of the received message.
- If we already have processed this don't do anything.
- Decrement the TTL. If TTL is zero after that don't relay it anymore. We are done.
- If we have not processed it we do the following :
  - Create a new invalidate_req. Add it to the ipending list.
  - Walk the list of peers and do the following :
    - call invalidate() RPC to that peer.

Oneway RPC Service : invalidate_1_svc:

This is a oneway RPC call used the peer servers to relay the invalidate calls to it peers. This calls the propagate_invalidate(). And uses the udpate_rec() to update the local record with the new version.

RPC Service : validate_1_svc:

This call is used by the pull method to check if a given file is out of sync. This is called by the cache-server to the origin server. The origin server of the file fetches the local of the file and returns it to the caller. The caller on receipt checks if the returned version is higher that the local record. If so, it updates it using update_rec().

Reaper Thread : reaper_thread() :

This is a thread is responsible for pruning the old queries on the linked list. The outline of the logic is :

- When a new request is added to the linked list we timestamp it and add it to the tail of the list.
- The reaper thread wakes up every 30 seconds and walks the list looking for nodes which are older than 60 secs. If so, it deletes them from the list.

We now have 2 two linked lists :

- ipending : This is the linked list of all the invalidate requests that are being processed or processed in the recent past.
- Qpending : This is the linked list of all the query requests that are being processed or processed in the recent past.

NOTE : The rest of the design and code remain the same as PA#2. Hence not describing it here.

Design of Peer client

Related source: obtain_client.c
Related executable : obtain_client – This the application that the user uses to fetch files.

Outline of the logic :

The client takes the name of the file and the index server as the arguments. The following are the steps that it takes :

- Queries the index server for the file that is requested for. The index server responds with a list of peers that are sharing that file.
- The client now asks the user to pick of the server (from those listed).

- It then fetches the file in chunks of 4Kbytes .
- ***Once the file is fetches it then calls the addcache() RPC call to the local server to add the file to the localcache. This is the modification to the code compared to PA#2.***

We have the following additional features :

- We have an additional option “-f” when specified the client fetches the file from the first peer listed. This is the non-interactive version which allows us to issue requests in a loop for stress testing.
- In case we are unable to fetch the file from the chosen peer it tries to fetch it from other listed peers.

Resiliance and Failure handling aspects :

The following are some of the resiliancy and fault handling aspects that we have incorporated :

1. Memory allocation failure : We make sure that we check that the allocation of memory has succeeded. And if there is a failure we handle it gracefully. For example, if we fail to allocate memory for a node to be added we decide to continue with other operations. This works because, eventually the reaper_thread() would kick in and make sure the unused and old entries are reaped to make sure more memory is available.
2. Failure handling : It is quite possible that a peer which had advertised that it can serve a file could be either down or may not be serving that file anymore. In such a case the client gracefully switches to the next node which is sharing the same file. Also, during relaying of the queries it is quite possible that the peer server is unreachable. In such a scenario we continue with relaying to other servers instead of failing the operations.
3. Restarting of peer server : We **donot** handle this case today. However, this can be easily done with a script which keeps sending NULL RPC calls to monitor the status of the server. And if the server does not respond it could be restarted.
4. Avoiding duplicate entries in the local cache : We make sure that we donot add duplicate entries of peers to the local cache.
5. Optimization is avoiding unwanted broadcasts : As explained under the section “RPC Call - search – outline of the logic” we have optimization is place to avoid broadcasting of queries to peers which we already know are sharing the file.
6. Framework for graceful handling of large number of peers : Today each reply for the query can hold a maximum of 16 peer names. However, we have designed the code such that if there are more peers serving the same file, multiple replies (b_hitquery() calls) are made to send them in chunks of 16.

Future work :

As described above we are facing the following issues :

1. With MAXCOUNT set to anything above 1 we are facing xdr errors during encoding (although this is the autogenerated code from rpcgen. This needs some fixing.
2. Reduce the number of calls generated because of the problem described above. Today we modified the search_1_svc() code to send back consecutive entries using the offset of the previous entry as the index for next calls.
3. Improve the stability of the server code.