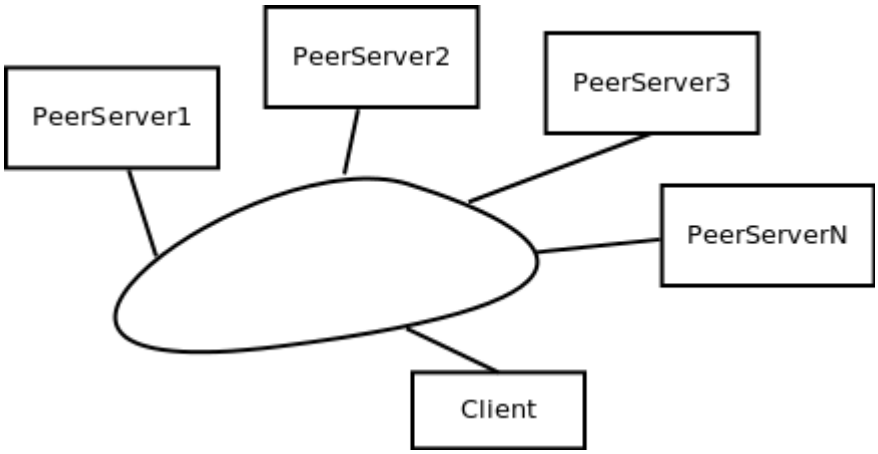
Design Document of "A Simple GNUtella Style Peer to Peer File Sharing System"

Abstract

This document describes the design of the various components of the 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.



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()

Querying, Relaying and Indexing Framework

For indexing purpose we decided to use a simple directory/file based database. The outline is the same as we described in PA#1.

Here is the outline of the framework.

The index server creates an index directory ("/tmp/indsvr") when the first request for registration is received. When a peer server tries to register a file with the index server it checks if a file by that name already exists under "/tmp/indsvr". If not it creates one and writes the name of the peer in that file. If the file was already registered by another peer we just append the name of the new peer to the file.

Note : The index server before adding the name of the peer check if the peername is already registered for that file. If it already is registered, it just returns success. This avoids having duplicate entries of peer names in the registry record (aka. the file in our case).

Example :

Let's assume we have a peer by name "peer1" which is trying to register the file "file1.txt" with the index server.

Steps taken by Index server :
- The index server will check if the file "/tmp/indsvr/file1" already exists.
- If not found it creates the file ("/tmp/indsvr/file1").
- It then writes the name of the peer server ("peer1") to the file.

The contents of the file "/tmp/indsvr/file1" would look as below :

```
-- snip --
$ cat /tmp/indsvr/file1
peer1
-- snip --
```

Now, if another peer by name "peer2" registers the same file "file1" here is what happens :
- The index server finds that the file "/tmp/indsvr/file1" already exists.
- It just appends the name of the peer ("peer2") to the file "/tmp/indsvr/file1".

The contents of the file "/tmp/indsvr/file1" would look as below :

```
-- snip --
$ cat /tmp/indsvr/file1
peer1
peer2
-- snip --
```

This model of indexing keeps the code simple and fairly scalable. Because, the indexing logic is now handled by the underlying filesystem's directory implementation. FIlesystems typically scale well with directory entries up tens of thousands. And this fits out bill.

However, if we wish to scale beyond this a database (like SQLite) can be easily plugged in.

Oneway RPC Calls :

In this implementation of peer serrver we need broadcast(relay) our query to all the clients and not wait for results. The responses would come in later in an asynchronous fashion. Hence, for this purpose the normal blocking RPC calls are not suitable. Hence we modified these to oneway RPC calls by changing the timeout parameter. (Refered to the book Power Programming with RPC by John Bloomer).

Registering the local files and localcache - outline of the logic

The peer server registers all the files that it plans to server to its local cache using the framework described above. add_peer() is the routine which implements this. This same interface/framework is used to cache the results from b_hitquery which will be described further down.

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];/* Name of the file that was asked by the app */
        int count;          /* Number of peers serving the file */
        char peers[BUFSIZE];/* List of peer names indexed at MAXHOSTNAME
length */
    };
```

For example if peer3 download app was looking for "file1" then the result returned by the index server would look as below :

```
    query_rec {
        char fname[MAXNAME] -> "file1"
        int count;          -> 2
        char peers[BUFSIZE]; -> ["peer1" "peer2"]
        int eof;             -> 1, if this is the complete list of peers
    };
```

However, if the requested file is not registered we return with count = 0 indicating that no peer is serving that file.

b_query_propagate – Helper routine to relay requests :

This is the helper routine used by search_1_svc and b_query_1_svc() to relay the query to all the peers if the server has not already processed this query. It uses a linked list 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 but do the following :
  - Look at the local cache and call b_hitquery() with the results to the uphost.
- If we have not processed it we do the following :
  - Create a new query_req. Add it to the pending list.
  - Walk the list of peers and do the following :
    - Check if the cached index file already has the name of the peer.
    - If the peer name is not in the cache send it a b_query() request.
  - Look at the local cache and call b_hitquery() with the results to the uphost.

Oneway RPC Service : b_query_1_svc

This is a oneway RPC call used the peer servers to relay the queries to the peers. Also, it returns the results of the localcache using b_hitquery_1() call to the upstream host. Following is the outline of the logic :

- propagate the query to the peers if we have not already processed this query (identified by the unique message id) – b_query_propagate() is the routine which implements this.
- Send back the results from the local cache to the upstream host by invoking the oneway RPC call b_hitquery_1() to upstream host.

Oneway RPC Service : b_hitquery_1_svc

This service is called by the peers to relay back the results as part of b_query() request. The following is outline of the processing done by this routine :

- Look for the msg_id in the pending list. If not found, the result came in late (beyond 1 min timeout) and possibly was reaped by the reaper thread. In such case we just record the results in the local cache.
- If the request was found on the pending list, find the upstream host id and relay back the response to it.
- Now add the results to our local cache to make sure that we can use them in future requests for the same file.
- In this above process if the uphost is same as the localhostname it means we have arrived at the source host which initiated the search in the first place. In such case it checks if this is the last response that the server was expecting. If yes, it wakes up the search_1_svc() by signalling the condition variable so that it can proceed to sending the response to the client.

RPC Call obtain() - Outline of the logic

This RPC call allows the peer client app to fetch a chunk of a file (4K) in our case.The following is the argument passed by the client :

```
struct struct request {
    filename name; -> The name of the file which chunk is requested
    int seek_bytes; -> The offset from which the chunk is requested
};
```

This request from the client asks for chunk of 4Kbytes from the offset seek_bytes.

The obtain_server opens the requested file, seeks to seek_bytes and then copies the chunk of 4Kbytes into the result. The result structure looks like :

```
struct datareceived {
    filedata data; -> The file data chunk returned
    int bytes; -> The actual number of bytes returned (max 4Kbytes)*/
};
```

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.

Multithreading the Peer server :

In order to enable the index server to serve multiple requests simultaneously we are creating a pthread for every request that comes in. Here is the outline of the modification that is done :

The svc_run() routine invokes the dispatcher routine registered by (svc_register()). The dispatcher routine does the following tasks :

1. Check the procedure number of the incoming RPC request and appropriately selects the input argument type and the result type. It also picks the appropriate service routine.
2. It then extracts the arguments from the svc_req pointer passed to it.
3. With the arguments and the holder for the results in place it calls the service routine. This is where the logic of registry or search is exercised.
4. It then sends the results back to the client.

To support multithreading we have modified the dispatcher routine (indsrvprog_1) as below :

- We allocate a new data structure of type tdata_t which has the following info in it:
  - Pointers to svc_req, SVCXPRT and pointer to the service routine .

    - argument and result types and the arguments passed by the client
    - memory to store the results
- After steps 1 and 2 instead of calling the service routine directly we spawn a new thread and pass the tdata_t was created in the prior steps. The new pthread calls the service routine in its context. It then sends the reply back to the client with the results.
- Frees the memory allocated to the tdata_t struct which was the place holder for arguments and results.

With this approach the svc_run() is free to accept newer requests while the new thread created simultaneously services the previously accepted request.

Further Improvements :
The above approach is a little inefficient as we create one new thread for every new request that comes in. And the thread is destroyed once the request is serviced.

Here the cost of pthread creation is incurred for incoming request. We could improved this by pre-creating a threadpool of slave threads and service queue. The new requests that come in could then be placed on the service queue and the slave threads can pick these requests and service them.

Locking stratergy for multithreading support :
In order to support multithreading the following care has been taken :

- The skeleton routines are generated by passing "-M" option to rpcgen. This ensures that the skeleton routines are reentrant.
- We use flock() on the files that are being accessed. We take the shared lock (LOCK_SH) for search() operation (we only read the contents). Where as the registry() operation takes an exclusive lock (LOCK_EX) on the file as it could potentially modify the file.
- The linked list has one lock which is taken while walking the list.
- Each node also has a lock which is locked when any routine is accessing the node.

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 .

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 : The obtain_client today reads only the first 16 results. However, with the current framework it would be very easy to modify obtain_client and search_1_svc() to return multiple packets of results. Note: We have already added the fields eof and count fields to the query_rec structure and count field to query_req structure. The code for search_1_svc() already populates eof and count of query_rec correctly.