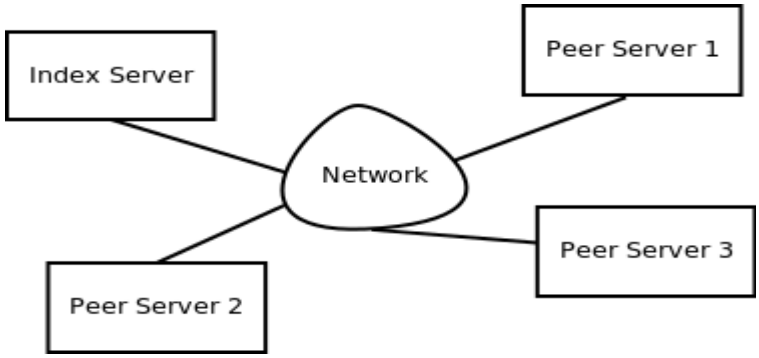
Design Document of "A Simple Napster Style Peer to Peer File Sharing System"

Abstract

This document describes the design of the various components of the Napster Style Peer-to-Peer File Sharing System. The main components of this are :

1. Index Server : This maintains the index of all the files shared by the peers in the system
2. Peer Server : This is the peer server which registers the files it wants to share with the index server and then serves the files when requested by the peers.
3. Peer Client : This is the client application which will query the Index server 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 Index Server :

Related Source files :
ind.x – The RPC description of the services
ind_xdr.c – XDR routines generated by rpcgen
ind_clnt.c – Client stubs generated by rpcgen
ind_client.c – Sample client code (handwritten to test indexing)
ind_server.c – Server logic which implements the actual service
ind_svc_mt.c – The Multithreaded code which handles multiple requests.
Makefile.ind – The Makefile to compile the below listed executables.

Related Executables :
ind_server – The index server which maintains the registry of files registered and allows querying.
ind_client – Test application to test indexing server.

The Index server is responsible for the following :
- keeping an index of all the files registered by the peer servers.
- allow peer servers to query for a particular file. Here the index server returns the list of peer-server names which are sharing that file.

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

Indexing Framework

For indexing purpose we decided to use a simple directory/file based database. 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.

RPC Call - Registry - outline of the logic

This RPC call allows the peers to register one file at a time with the index server. The following are the arguments that are passed by the peer server :

```
struct registry_rec {
     string  peer<MAXNAME>; /* The name of the peer server */
     string  fname<MAXNAME>;/* The filename that it wants to register */
     int     ret;
};
```

The index server on receipt of this request registers the name of the peer as described in the section "Indexing Framework".

RPC Call - search – outline of the logic

This RPC call allows the peer download application to query the index 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 index server 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"]
    };
```

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

Multithreading the Index 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 in Index server 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.

Design of Peer Server (obtain_server) :

Related Source Files :
obtain.x – The RPC definitaion of the peer server
obtain_xdr.h – XDR routines generated by rpcgen
obtain_clnt.c – The client stubs generated by rpcgen
obtain_client.c – The client app which first queries the index server for a file and the contacts the peer server for the file.
obtain_server.c – The server logic which serves the chunks of files requested.
obtain_svc_mt.c – The multithreaded server code which registers the files to be served to the index server and serves the files by calling into service routines from obtain_server.c

Related Executables :
obtain_server – This is the peer server which is executed by every peer.
obtain_client – The obtain application that the user uses to fetch the files.

The obtain server just supports one RPC call obtain(). It takes the directory which the peer wants to serve and the index server's name. The server first registers with index server all the files under the directory that is being shared. It then calls svc_run() to serve the requests from 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)*/
};
```

Multithreading of Obtain Server :
The multithreading of the Obtain server is the same as that for the Index server.

Locking stratergy in Obtain Server :

The obtain server need not take care of any locking for multithreading because every request issued by the client is independent and complete. Also, since it only reads the chunks and does not modify anything there is no inconsistency because of multiple threads accessing the same files.

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.