

Programming Assignment 2 – p2psearch

Due date: November 8, 2005 (Assigned on October 11)

Extension is granted through November 11, 2005 for all who may need it. The last (3rd) programming assignment is likely to be distributed prior to this date, however.

1 Objectives

This programming assignment is about building a peer to peer **searching** functionality that will be used orthogonal to your implementation of `richftp` from the previous assignment. The last programming assignment will deal with merging the search and ftp.

Since it is infeasible to run a thousand peers over a real network, we are providing you with an implementation of an **emulated network** layer (EmulNet). Your p2p search implementation will sit above EmulNet, but below an App layer. Think of this like a 3 layer protocol stack with App, P2P search, and EmulNet as the three layers (from top to bottom). More details are below.

2 The Three Layers

The three layer implementation framework we are providing will allow you to run multiple copies of a peer search implementation within **one process running a single-threaded simulation engine**. Here is how the three layers work.

2.1 Emulated Network: EmulNet

EmulNet provides the following functions that your P2P layer above it should use.

- `void *ENinit(struct address *myaddr, short port, char *joinaddr);`
- `int ENp2psend(struct address *myaddr, struct address *addr, char *data, int size);`
- `int ENrecv(struct address *myaddr, int (* enqueue)(void *, char *, int), struct timeval *t, int times, void *env);`
- `int ENcleanup();`

ENinit is called once by each node (peer) to initialize its own address (myaddr). ENp2psend and ENrecv are called by a peer respectively to send and receive waiting messages. ENrecv enqueues a received message using a function specified through a pointer enqueue(). The third and fourth parameters (t and times) are unused for now. You can assume that ENsend and ENrecv are reliable. ENcleanup is called at the end of the simulator run to clean up the EmulNet implementation.

These functions are provided so that they can later be easily mapped onto implementations that use TCP sockets. To facilitate this in the future, calls to the above functions are made respectively through function pointers MPinit, MPp2psend, MPrecv, MPcleanup.

Please do not modify the files `emulnet.{c,h}` given to you. We may replace it with our own implementations during testing. You should only use the above functions to access the EmulNet layer, and should not access the EmulNet data structures directly.

2.2 Application: App

This layer drives the simulation. Files `app.{c,h}` contain code for this.

Look at the `main()` function. This runs in synchronous periods (`globaltime` variable). During each period, some peers may be started up, and some caused to crash-stop. Most importantly, for each peer that is alive, the function `nodeloop()` is called. `nodeloop()` is implemented in the P2P Search layer and basically receives all messages that were sent for this peer in the last period, as well as checks whether the application has any new waiting requests.

The App layer exports two functions:

- `void PollApp(member *node);`
- `void App_flkupcllbk(void *env, char *fname, int fnsz, address *ansaddr, enum Lkuptype lkuptype);`

`PollApp` is called from `nodeloop()` (P2P search layer) in order to query the App layer for new waiting requests. When a request is completed, the P2P search layer initiates an upcall into the App Layer through `App_flkupcllbk()` to return the answer (`ansaddr`). `ansaddr` is `NULL` if there is no answer, and otherwise is the peer address. Peer addresses are 6 bytes (which will accommodate IP address+port number in the next programming assignment).

2.3 P2P Search

The most important function in this layer is called by the App layer in order to request a file operation:

- `enum Lkupstatus fileops(member *node, char *fname, int fnsz, void (*cllbk)(void *, char *, int, address *, int), enum Lkuptype lkuptype)`

The `lkuptype` parameter is either `OUTS_INSERT`, `OUTS_DELETE`, or `OUTS_LOOKUP`. The first two parameter types simply cause creation or deletion of metadata about the file that is being inserted or deleted from **this** node. Read that again. Only `*metadata*` (i.e. filename, size, etc), rather than the actual file will be stored in the `p2psearch` overlay. The file will only reside on the uploading peer. In the next programming assignment, you will use `p2psearch` for finding the peer that holds the file, and then transfer the actual file from that peer using `richftp`. For now, you will only deal with insertion, lookup, and deletion of metadata.

The last parameter (`OUTS_LOOKUP`) specifies that a lookup is requested for the filename.

The return answer for these three operations (`ansaddr` in `App_flkupcllbk()`) should be:

- `OUTS_INSERT`: peer address on success, `NULL` on failure.
- `OUTS_DELETE`: peer address on success, `NULL` on failure.
- `OUTS_LOOKUP`: peer address on success, `NULL` on failure.

You may assume that filenames are unique throughout the system, that is no two files inserted have identical names.

3 What Does the Code Do Currently?

As given to you, the code prints out debugging messages into `dbg.log` (format is `node_address [globaltime] message`). You can turn debugging on or off by commenting out the `#DEFINE DEBUGLOG` in `stdincludes.h`.

Two message types are currently defined for the P2P search layer (`mp2_node.c` implementation) - `JOINREQ` and `JOINREP`. Currently, `JOINREQ` messages are received by the introducer. The introducer is the first peer to join the system (either `0.0.0.1:0` or `1.0.0.0:0`, depending on the endian-ness of the system you are running on).

The best place to start your implementation is to have the introducer reply to a `JOINREQ` with a `JOINREP` message. The next section lists all the functionalities you have to implement.

4 What Do I Implement?

Most of your code will go into the P2P search functionality in file `mp2_node.{c,h}`. You will need to of course implement `fileops()`, but before that you will need to implement several other functions as a part of `nodeloopops()`, and the `Process_*`() functions. These are loosely described below, which means you have considerable flexibility in choosing implementation details, **except that your final submission should work with alternative implementations for `emulnet.{c,h}` and `app.c`**. We will use these alternative implementations for automatic testing.

We will be implementing a `p2psearch` overlay. Here are the functionalities your implementation must have:

- **Introduction:** Each new peer contacts a well-known peer (the introducer) to join the group. This is implemented through `JOINREQ` and `JOINREP` messages. Currently, `JOINREQ` messages reach the introducer, but `JOINREP` messages are not implemented. `JOINREP` messages should specify a maximum of 10 peers (as neighbors for the new peer). The introducer does not need to maintain a list of all peers currently in the system; a partial list of size at most 10 should be maintained.
- **Membership:** Each peer should have at most 10 other peers as neighbors. These neighbors can be chosen in any way you wish (random choice is fine), but they will need to be updated should any of the peers fail. When a neighbor fails, this will need to be detected, and a new peer included as neighbor in its place. You may wish to do this with a Ping-Pong style protocol, and a heartbeating protocol between neighbors. Notice that the heartbeating protocol between neighbors may be needed to time out failed peers.
- **Search:** Implement a search procedure that is different from a pure depth-first procedure. You are free to parameterize your search procedure (e.g., a parameter may specify the maximum number of times a request is forwarded). Do choose these parameters such that the search completes in within a finite time.

Some of the things that you will probably need to modify are the struct member in `mp2_node.h`, and the enum `Msgtypes` in `msgheader.h`. The latter will be needed as you add in new message types.

Optionally, you are free to optimize the performance of your system by spreading metadata about files farther out (instead of maintaining them only at the file's homenode) so that lookups are more efficient.

Do describe all your design decisions clearly in the `README` file.

5 What are These Other Files?

debug.{c,h} has a LOG() function that prints out stuff into a file named dbg.log. You can turn debugging on or off by commenting out the #DEFINE DEBUGLOG in stdincludes.h.

params.{c,h} contains the setparams() function that initializes several parameters at the simulator start, including the number of peers in the system (EN_gpsz), the rate at which these peers join starting from time $t = 0$ (STEPRATE), and the global time variable globaltime.

queue.{c,h} has an implementation of a queue **that you should not modify**.

The remaining files nodeaddr.h, requests.h, MPtemplate.h list some necessary definitions and declarations. You may want to avoid modifying these files.

Why is the Code Structure So Involved? There are two reasons.

Firstly, think about the issues involved in converting this into a real application. All EN*() functions can be easily replaced with a different set that sends and receives messages through sockets. Then, once the periodic functionalities (e.g., nodeloop()) are replaced with a thread that wakes up periodically, and appropriate conversions are made for calling the other functions node_start() and rcvloop(), your P2P search implementation can be made to run over a real network!

Secondly, this structure allows us to debug (and even measure the performance through traces) of a peer to peer search protocol easily and on a single host machine. Compare this with the debugging challenge for several hundred processes running on a real network. Once the simulation engine works, you can convert the implementation easily into one for a real network, and it will work.

6 What Do I Turn In?

6.1 Testing

We will test whether your overlay allows peers to join correctly, recovers on peer failures, and whether file insertions/deletions/lookups happen correctly. Make sure your system works with up to 1000 peers in the system (EN_gpsz=1000). Make sure your code is commented enough for us to understand it.

You should describe the design decisions of your protocol in the README file, and this will be graded too.

6.2 Submitting

Your code should compile with the Makefile that you submit with the command “make”, and should produce an executable “app” in the same directory. You should compile all your code with the -Wall flag, but your code should not display any warning messages during the compilation. Also, put all of your printf statements inside of # DEFINE DEBUGLOG ... # ENDIF. Include enough comments inside of your code for us to understand it, and describe the internals of your protocol in the README file.

Your code will be tested automatically by replacing the files emulnet.c, emulnet.h, app.c and app.h with our implementation. So, you should ensure before handing in that your code compiles and runs with different implementations of app.c. The sample files are not meant to serve in place of testing your code, and testing only with the samples is unlikely to find all bugs. Your code will be tested more thoroughly.