The P242P File Sharing System
CSC 360 Fall 2011

The goal of this project is to obtain experience with *designing* and *implementing* complex systems in a team. In this case, the complex system is a **peer-to-peer** (P2P) **file sharing system**. We will get this running on the machines in ECS 242, and then move on to WestGrid!

Background and Setup
In mid-2005, the United States Supreme Court unanimously decided that the defendant P2P file sharing companies, Grokster and Streamcast (maker of Morpheus), could be sued for inducing copyright infringement. That's food for thought(!), so be careful, but in this assignment you will build a file system based on a bare bones *gnutella v0.4*. In your P2P system, participants can come-and-go, and there is no single point of failure. Every client is a server and every server is a client (a "servent"(?!)). To test your system, you can use processes (communicating through sockets) on the same host, or you can use a subset of the machines in ECS 242, listed below.

142.104.72.2	u-arch.csc *(instructor workstation)*	142.104.72.18	u-mandriva.csc
142.104.72.3	u-centos.csc	142.104.72.19	u-mepis.csc
142.104.72.4	u-debian.csc	142.104.72.20	u-pc-bsd.csc
142.104.72.5	u-devil.csc	142.104.72.21	u-puppy.csc
142.104.72.6	u-elive.csc	142.104.72.22	u-red.csc
142.104.72.7	u-fedora.csc	142.104.72.23	u-slackware.csc
142.104.72.8	u-foresight.csc	142.104.72.24	u-slax.csc
142.104.72.9	u-freebsd.csc	142.104.72.25	u-suse.csc
142.104.72.10	u-gentoo.csc	142.104.72.26	u-turbo.csc
142.104.72.11	u-kanotix.csc	142.104.72.27	u-ubuntu.csc
142.104.72.12	u-knoppix.csc	142.104.72.28	u-vector.csc
142.104.72.13	u-kororaa.csc	142.104.72.29	u-vlos.csc
142.104.72.14	u-kubuntu.csc	142.104.72.30	u-wolvix.csc
142.104.72.15	u-linspire.csc	142.104.72.31	u-xandros.csc
142.104.72.16	u-linux.csc	142.104.72.32	u-xubuntu.csc
142.104.72.17	u-lunar.csc		

Step 1: Your Disk... It is just a FILE! (A *Fisk*?) Due in 1 week... Nov 4

A file system must be able to store its data in a persistent manner. You first must design a simple disk library that reads and writes 1-Kbyte disk blocks. Your P242P file sharing system will be built, eventually, on top of this interface. Your disk library will use a regular single Unix file to emulate a disk. At this lowest level in the system, blocks will be written using a *block number*. You will use this number to compute the offset into the Unix file. For instance, disk block 10 will be at byte offset `10 * 1024 = 10240`. Use *lseek* to simulate your disk activity.

The disk library's interface is as follows:

```
int openDisk(char *filename, int nbytes);
int readBlock(int disk, int blocknr, void *block);
int writeBlock(int disk, int blocknr, void *block);
void syncDisk();
```

- **openDisk** opens a file (filename) for reading and writing and returns a descriptor used to access the disk. The file size is fixed at nbytes. If filename does not already exist, openDisk creates it.
- **readBlock** reads disk block blocknr from the disk pointed to by disk into a buffer pointed to by block.
- **writeBlock** writes the data in block to disk block blocknr pointed to by disk.
- **syncDisk** forces all outstanding writes to disk.

The calls return an error if the underlying Unix system calls (to access your file) fail.

Step 2: The Local File System (Due in 2 weeks) Due Nov 14

Once you have persistent storage (your *fisk*!), you are ready to implement the bare bones of a file system using your disk library. There are two main challenges in implementing this local filesystem: (1) the main-memory and on-disk data structures to map files names onto disk blocks and to keep track of free blocks; and (2) the disk layout.

Three of the critical functions provided by a file system are:

1. **Naming**: names must be mapped to the disk blocks that they are associated with.
2. **Organization**. Files are typically stored in a directory.
3. **Persistence**. Data is entrusted to the file system and is expected to remain there (uncorrupted and undeleted) until an explicit request is made to delete it. File systems also provide mechanisms to modify existing files.

There are three main data structures file systems manage:

1. *A disk block freelist*. This is a simple data structure (for instance, a bitmap) that tracks what blocks on the disk have been allocated. This data structure is, for obvious reasons, persistent (i.e., on disk). You will have to establish some convention for locating it across reboots.
2. *Inodes*. An inode is persistent memory that contains pointers to disk blocks or to ``indirect blocks'' (you can think of these as other inodes). Inodes are used to map a file to the disk representation of a file: each file has its own inode, the inode has pointers to all disk blocks that make up that file.
3. *Directorie*s. Directories are persistent mappings of names to inode numbers for all the files and directories that the directory is responsible for.

You should provide functionality for very basic file operations (create, open, read, write, sync, close). Additionally, your system should be able to run, be powered down, and then be ``rebooted'' in the state that it was in when it was shutdown (i.e., all files that were written at shutdown time should still be there, and no garbage should suddenly appear).

Failures? They happen! Be ready… writing to a file in your system will involve a number of steps: allocating space for the file on disk, modifying the file's inode, modifying the disk block freelist, and finally, writing the data to memory. Failure after any of these should not corrupt the system (e.g., will not lose disk blocks, delete existing disk blocks, lose inodes, etc.). You should define a non-trivial (i.e., useful) recovery model that balances the utility of consistent state on disk with the expense of synchronous disk updates. You may require that a program be run at boot time to reconstruct parts of P242P---it should always be able to be recovered to a usable state.

Failures to guard against:

- **Use of allocated disk blocks**. Blocks may have been allocated to a particular file but the freelist was only modified in memory, not on disk.
- **Loss of disk blocks**. Blocks may have been removed from the freelist but not yet had an inode persistently modified to point to them.
- **Loss of data**. Data that has been written to disk should stay there.

There are, of course, many other failures possible, but these are the ones we will test.

Step 3: File Sharing—Getting Gnutty! (Due Dec 1!)

This is definitely a significant amount of functionality to have to work on for an assignment! Our intentions here are to keep things quite simple, and expose you to client/server communications in a very introductory way! There are several good online tutorials, examples and overviews, for structuring the communication aspects of your servents such as http://beej.us/guide/bgnet/output/html/multipage/clientserver.html.

Descriptor	*Description*
Ping XXYYZZ	Used to actively discover hosts on the network. A servent receiving a Ping invocation is expected to respond with a set of one or more Pong descriptors. XXYYZ uniquely identifies your "version" of the protocol.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent. Only reply if PING includes unique XXYYZZ.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. The QueryHit data structure provides the recipient with enough information to acquire the data matching the corresponding Query.

You can consider this list as the basis for an API as well as message format. In this assignment, a servent connects itself to the network by establishing a connection with only one or two servents currently on the network (via the *"Ping"* operation). We will accomplish this in this by recognizing that only the *ECS242* machines are going to be used. Whenever a new servent comes on the network, it should randomly iterate through the candidate machines, using the Gnutella

"Ping" operation until it finds one or two (your choice) peers running your software. Note that while it can only *directly* interact with at most *two* of its peers, it must be able to find and retrieve data from *any* of the machines running your client.

When a servent receives a *"Query"* operation, it checks if it has files that match the query locally, in the directory that it's serving up (through the implementation of your local file system!). If the "time to live" value (*TTL*) is greater than 0, the servent decrements the TTL and passes the query on to one of its peers (identified when the servent connected itself to the network). The peer response is sent to the servent issuing the request, not directly to the client that originated the request. The servent issuing the request can either deliver the result incrementally to the client or all at once (this is a design decision).

Bonus marks! [10%] Real Gnuttiness!

Your system should be able to tolerate the following:

1. Machines killed arbitrarily (temporarily).
2. Servents killed arbitrarily.
3. Servents that might appear to implement the protocol, but are actually faulty or malicious.

These might be new, standalone servents, or they might replace your existing servents.

Grading:

Each program must be properly and reasonably completely designed, documented, (at least partially) implemented, and tested/evaluated. It is a lot of work, so it would be best to do this in **groups of 4**!

Though you are not expected to test this, you goal is to create a system that *could* scale to hundreds of machines, and the virtues of the system will be assessed in your attempts to address these issues in your *design*:

1. Maximize the hit rate of queries by using your servents.
2. Minimize network traffic.
3. Minimize the amount of disk space you are using (i.e., no "unnecessary" replication).
4. Minimize response time, both for queries and for getting files.
5. Handle the gnuttiness (bonus).

The grading for this part of the assignment will be:

1. 75%: The quality of the code itself. This includes: the readability of the code, the correctness of the code, the efficiency of the code, etc. (this will roughly break down into 25% for each of the disk, file system, and file sharing layers).
2. 25%: The quality of a short design document (at most 5 pages), containing: the basic design, implementation issues/challenges, known difficulties with your program (known cases where it does not work), testing methodology, analysis and improvements you would make.

MORE ON THIS SOON!!!!!