E&CE 454/750-4: Spring 2012
Project 1
Due: 11:59 PM Thursday 7th June 2012

For this project you are required to implement a crude version of the BitTorrent protocol for the purpose of automated file replication. BitTorrent normally has a tracker and is demand-driven. Instead, we will be using the protocol to allow a set of nodes to replicate files that are received at any given node within the set. To avoid the need to create a user interface, parse user requests, *etc.*, we instead define a simple four-command API:

```
class Peer {
  public:
    int insert(String filename);
    int query(Status status);
    int join();
    int leave();
}
```

This class, together with the Status class, Peers class, and any other necessary supporting classes will be provided in `.h` files on the course website. The `int` returned by each method is a return code, defined as follows: if the method is successful, then it returns 0. If the method has an error, it returns a negative code, with codes defined in the online header file, though you may wish to add to the online set. If the method has a warning, but is able to execute, then a positive integer is returned. Again, some codes are defined in the online header file, though you may wish to augment that set.

The methods are defined as follows. The `insert()` method takes a string which is expected to be a file of name `filename` and add it to the local peer. If the file does not exist or cannot be read, then the method returns a negative return code. If the file can be read, then it is stored with the other files that are being kept by this peer; once that storage has taken place, the method returns a 0. However, this is still only a local copy. The local peer then pushes the file out to the other peers using the BitTorrent approach of dividing the file into chunks and having the peers each receive chunks either from the originating peer or from other peers that may have received particular chunks already. The chunksize is defined by the constant `CHUNKSIZE` which is in the header file on the course website. It is up to you to define, as part of your solution to this project, what low-level socket protocol you will use between peers to satisfy this requirement. You will be expected to document your design.

The `query()` method is used to query the state of the files in the local peer. It receives an object of type Status and populates it with the necessary information to describe four parameters for each file:

1. The fraction of the file that is available locally
2. The fraction of the file that is available in the system
3. The least replication level
4. The weighted least-replication level

The `join()` method accepts a container of Peers and attempts to join the set; in doing so, it must push any files that it has locally to the set, as well as receive any that are in the set for which it has no copy.

The `leave()` method leaves the set of peers in which it is currently a member. In doing so it should close all sockets cleanly, and inform all peers that it is leaving. It is desirable that when a peer leaves, if it has a few unique file chunks that those be pushed out before leaving, but this can only be expected to be performed if there are a very small set of such chunks. It is more realistic to follow the BitTorrent approach of ensuring that chunks are are not heavily replicated are shared first.

Alternate Approach

It is desirable for systems such as this to messages received over a socket rather than being specified by a C++ API. To this end, the project will allow the design of such a protocol, and some suggestions in that direction, together with specifications for the protocol, will be given during the tutorial and in the online site.

Requirements

You are required to implement this file-replication system as described. You can use any language you like to implement it, but it must either work with the C++ API or respond to messages over a protocol, per the tutorial discussion and online website specification.

The user manual is too similar to this specification, and therefore of little value. In addition we will perform testing, so there is little point in doing a test document. However, short of reading your code in detail, determining how you implemented the system is non-trivial. You are therefore required to write a system manual describing how your system is designed and implemented. At a bare minimum it should describe the messaging protocols that you designed. If you do not know what a system manual should look like, imagine that you had to maintain the code that you are writing, but that you did not write the code. Someone else did. What would you need to know about the code in order to maintain it? This is what you should describe.

If you take the protocol approach, the protocol must be completely specified, together with sample usage.

Submission will be *via* Coursebook. The system is expected to run in the ecelinux environment.