CSG389 **Homework 4** **Assigned: 3/30/2005**
Spring 2005 CCS, Northeastern University **Due: 4/6/2005**

Problem 1 – A bunch of distributed file systems questions.

In this problem you will explore the strengths (and weaknesses) of two commercially successful distributed file systems – AFS and NFS. Specifically, you will look at the implications of state (or lack thereof) at the server when several clients are writing to the same file.

Recalling from the lecture the semantics of NFS server (the original version, i.e., no leases), it does not know when a client opens a file for writing nor when the client is done writing and closes the file. Because of that, NFS requires that the client writes thorough all changes i.e., every time a client writes a byte to the file, the client sends the appropriate block (e.g., 4 KB) to the server that writes it synchronously to the disk.

The AFS server, on the other hand, keeps a list of all clients that have a file open. The client modifies a local copy of a file and only when it closes the file are the dirty blocks sent to the server. The server then looks up all the clients that have a local copy of the same file and sends them an invalidation message via a callback. This message lists all the blocks that have been modified. The clients have to contact the server if they want to get the latest version of the data.

Assume a sequence of system calls (listed below) at four clients that all open the same file stored on a file server. The file is initially empty. The time increases such that $t_1 < t_2 < \ldots < t_9$. The syscall `write(fd,`*data*`)` means that the *data* is written to the file whose descriptor is `fd`. Answer the following questions. Give all scenarios whenever possible and briefly explain your answer.

Time	Client 1	Client 2	Client 3	Client 4
t_1	`fd=open("f1")`	`fd=open("f1")`	`fd=open("f1")`	`fd=open("f1")`
t_2				`write(fd,4)`
t_3		`write(fd,2)`		
t_4	`write(fd,1)`	`close(fd)`		
t_5			`write(fd,3)`	
t_6	`close(fd)`		`close(fd)`	
t_7				
t_8				`close(fd)`
t_9				

(a) Assuming file `f1` is stored on an NFS server, what is its content at t_9?

(b) Assuming file `f1` is stored on an AFS server, what is its content at t_9?

Assume a network failure occurred between t_6 and t_7 and it never recovered(i.e., the `close` system call failed on client 4).

(c) What is the content of the file at t_9 on an NFS server?

(d) What is its content on an AFS server?

Assume a network failure occurred between t_4 and t_5 and it never recovered (i.e., the `write` failed on client 3 and `close` failed on clients 1, 2, and 4).

(e) What is the content of the file at t_9 on an NFS server?

(f) What is its content on an AFS server?

Now assume that the server went down between t_4 and t_5, but everything else was functioning well. It took sufficiently long for the server to come back up so that all the clients finished their applications (since they didn't handle a failed `write` and `close` system calls) happily believing their data was saved.

(g) What is the content of the file when the NFS server came finally up?

(h) When the AFS server came finally up?

(i) Hopefully, you have noticed the different contents of the file resulting from failures occurring at different times. Which semantics do you think are "better" – AFS or NFS? Of course, it depends what "better" means. So define your metric of goodness and justify your answer. We want you to think about the tradeoffs that the designers had to face when they were designing the distributed file systems. In answering this question think how often this situation arises.

Let's look at scalability of the two protocols. Assume that there are 250 clients each running the following code:

```
int fd = open("file",O_CREAT);
for( i=0 ; i<1000 ; i++ ) {
    sprintf(string,"%d\n",i);
    write(fd,string,strlen(string));
}
close(fd);
```

Both file systems send 4 KB blocks. The size of a single RPC message is 256 bytes.

(j) How many messages and how many KB of data are handled by the NFS server?

(k) An AFS server?

Finally, a few more random questions:

(l) Do two NFS servers provide a common, global name space? Explain how or why not.

(m) Define *idempotency*. Why is it a useful property? For a distributed file system, give an example of an idempotent operation and a nonidempotent operation.

(n) What is the function of an AFS callback? Why isn't there an "NFS callback"? What would be the advantage of designing one into the NFS protocol?

(o) In general, access to local disks is faster than access to remote disks. Why bother with all the overhead of network attached storage? E.g., what's the "big win"?

Problem 2 – Content Delivery Networks.

Content delivery networks (CDN) or cooperative caching networks are a form of a storage system. Designed for the World-Wide Web, they improve performance for frequently requested web pages by form of replication and caching. In this problem, we explore the design decisions that parallel the decisions in a distributed storage (file) system. A web page is in essence an object just like a file. When an object's content is changed, the outdated versions have to be invalidated and new replicas propagated to the set of cooperating caches. For this problem, we assume static pages without server includes, PHP, or other dynamically created content. In another words, we assume that a page content changes on the order of minutes or hours rather than with every upload or second.

Last night you had a brilliant, marketable idea: a product for caching static web content in proxy servers. Your new service will be a distributed cooperative caching network—that is, you'll set up a collection of proxy caches that communicate with each other to share hot objects.

(a) Your first major design decision is to either **cache near the content** (placing caches close to the major Internet backbones and connection points) or **cache near the clients** (putting one cache at each of your client's sites). What are the advantages and disadvantages of each? Which would you choose?

(b) Your proxy will have a large memory (1 GB) for caching "hot" pages, plus disk space (100 GB) for caching "warm" pages. There are several ways to store objects on disk for rapid lookup:

  (1) You could create N cache directories (where N is large) and always place new objects in the emptiest directory, keeping a "object-to-directory" lookup table on disk.

  (2) You could create N cache directories and hash each object name to a directory.

  (3) You could create a directory structure based on object names. For example, the URL "http://host.subdomain.domain.tld/dir1/obj2.html" would go in `/cachedir/tld/domain/subdomain/host/dir1/obj2.html`.

  (4) You could create a single cache directory and store all objects there based on their full object name (converting "/" to "-", for example).

Compare each of these schemes in the areas of (i) number of disk accesses needed to access an object; (ii) ability to co-locate related objects (e.g., a web page and the JPEG images on that page); (iii) "fair sizing" among directories (e.g., all directories are about the same size). Choose any file system that you've learned about this semester and base your answers on that file system.

(c) Why might it be useful to relax consistency requirements for the disks on your web proxies—that is, *not* guarantee that data are in a recoverable state on disk after a crash? Would you recommend doing this for web *servers* for the same reasons? Why or why not?

(d) When the disk is very busy, and you run out of memory to store "hot" pages, you can free memory by **throwing out "unwritten" objects** (not yet written to disk) or **throwing out "written" objects** (already on the disk). What are the advantages and disadvantages of each? (E.g., which scheme reduces the load on the disk, etc.?)