Project #3: Distributed File System

Due: May 19, 2006 11:59pm

Overview

For this assignment you will implement a simple distributed file system with client-side caching. This system will function in a manner similar to NFS.

Though there are some requirements discussed below, and a specific abstraction that must be provided on the client, the design of the system is primarily up to you. In particular, you are free to use a distributed object system (e.g., CORBA or Java RMI) or sockets for the communication between clients and the server.

Part 1: Cacheless System

For this project, each file will be accessible to all and allow both read and write access. The server will store all files in the system in a single directory named `FILES`. Subdirectories need not be supported. Access to files will be through unique file ids.

Server

The Server, named `DFSServer`, must provide the following primitive operations to clients.

- create: Allow a client to create a file in the system and obtain the unique file id (UFID).
- lookup: Allow a client to obtain the unique file id (UFID) for a file in the system.
- read: Allow clients to read some number of bytes starting at a given offset from the specified file.
- write: Allow clients to write some number of bytes starting at a given offset from the specified file.
- get time (Part 2): Allow clients to retrieve the time at which a file was last updated.

The server must support multiple simultaneous connections. Moreover, though writes to a single file must be serialized, writes to separate files are allowed to occur concurrently.

Client Module

The Client Module, `DFSClientModule`, provides the system level interface to the server. This module performs the actual communication with the file server. This module is, in essence, a bridge between the interface discussed below and the actual file server. All operations provided directly to the user (through the interface discussed below) will be routed to the file server through this module.

The client module must support the following static operations (the cache related operations are for Part 2 of the project). The first three may be invoked directly by a client application; the remainder are intended for use by only the client interface defined below.

- initialize: This method must be invoked prior to any use of the file system by the threads in the current virtual machine. This method takes, in this order, the server name (`String`), the server port (`int`), a cache chunk size (in bytes, `int`), and an initial file refresh interval (in seconds, `int`).
- setRefreshInterval: Sets the file refresh interval to the specified integer number of seconds.
- read
- write
- create
- lookup

Client Interface

The Client interface to the distributed file system is defined through implementations of standard `java.io` classes (if you are not using Java, then you must map to analogous features in your implementation language). For this project, you must implement three classes: one input stream and two output streams. The two output streams differ in their interactions with the client-side cache (see below), but are both described here.

DFSInputStream

The `DFSInputStream` implements the functionality described in (and is a subclass of) `java.io.InputStream`.

This implementation is supported by the Client Module described above. Your specific implementation must, at a minimum, define the `read` methods from `java.io.InputStream`. The constructor takes the name of the file from which to read as its single `String` argument.

CommitOutputStream
The `CommitOutputStream` implements `java.io.OutputStream` by interacting with the Client Module. You must implement the `write`, `close`, and `flush` methods. For this output stream, any execution of `close` or `flush` will cause the "dirty" cached portions of the file to be sent to the file server.

WriteThroughOutputStream
The `WriteThroughOutputStream` implements `java.io.OutputStream` by interacting with the Client Module. You must implement the `write` methods. On each write, both the local cached copy and the server copy are updated.

`Note: InputStream` and `OutputStream` define variants of the `read` and `write` methods that work on arrays of bytes. The default implementations invoke the single byte operation for each byte in the array. For synchronization purposes, your implementations will need to pass the entire array to the client module.

You *can* test your system at this point, without concern for caching discussed below. The `OutputStream`s won't differ in behavior, but instead will immediately write to the server.

Part 2: Caching

In a distributed file system, one would typically implement the client module at the OS level. We do not have this luxury, nor is it necessary to illustrate the topic. Instead, for this project, we will view the virtual machine as the machine itself. As such, the client module will provide the distributed access for the entire virtual machine (each thread is a client of this module).

To address performance issues, the client module will cache files for the local virtual machine. The files will be cached in chunks. Only those chunks that are actually used will be stored in the cache (i.e., if only the first few bytes of a file are accessed, then only the chunks that correspond to those bytes will be cached). The chunk size is set when the client module is initialized (the last chunk may be smaller than the chunk size).

All writes, once the chunk has been read and stored locally, will be performed directly on the chunk without checking the server for updated information (the write-back policy is defined above).

Each read is performed on the corresponding chunk. A read, however, must check that the lease on a chunk has not expired. Specifically, the client module must maintain a timestamp for each chunk (this may be independent of the timestamp in the local file system). This timestamp represents the last time the client module requested the file update time from the server (the server will maintain a single update time for the entire file). Before each read, the client module must check that the current time is less than the timestamp for the appropriate chunk(s) plus the refresh interval (i.e., if the current time is T_c, the timestamp is T_f, and the refresh interval is r, then $T_c < T_f + r$).

If the lease has expired, then the client must obtain the file update time from the server and, if the file has been updated since the last check, refresh the cached copy by reading the appropriate data from the server.

Your client must allow concurrent access to different files, but should serialize access to the same file. Since a single read/write might access multiple chunks, it is acceptable to provide a thread exclusive access to the entire set of chunks for a file.

Handin

Submit your source files using handin on hornet/falcon. To do this, execute

handin akeen proj3 <your files here>

Grading will be divided as follows.

Part	Percentage
1	70
2	30