A Prototype for a Stateless File Server and its Cache Mechanisms

This project is adapted from a project designed by Dr. Kohei Honda, Queen Mary University of London

1. Introduction

This project concerns the design and implementation of a simple stateless file server and its client-side API. It will give you experience in the use of TCP socket for communication between client and server and in the design of a client-side cache.

2. Basic Description of Project

Background. This project is about the design of the system components required to support an application which reads and writes a file over the Internet.

Objectives. Your task is to make a pair of components: a file server (to be remotely situated) and a client-side library. Clients can use these two components to access a file located remotely, by means of TCP over the Internet. The objective is for you to obtain a deeper understanding of stateless file servers and their cache mechanisms. You are asked to produce two pieces of software:

(1) A simple stateless file server, which reads from files and writes to files in response to requests transmitted over TCP. It provides *lookup*, *read*, *write*, and *getattribute* operations. You do not need to provide other mechanisms such as security, locking, concurrency control, and fault tolerance.

(2) A library class (API) for a client of (1), enabling it to interact with (1) over the Internet. This part is to be done in two stages. In the first stage no caching is required. In the second stage you should incorporate caching.

You are *not* required to implement a file server's most complicated bits, i.e. the direct interface with a disk device driver. Your server simply uses Java's standard API to read and write files. This is because the objective of this project is to create a stateless file server which allows clients to access files via TCP requests, and to learn about cache mechanisms on the client side.

Stages. There are two stages in this project. The first stage is to design and implement a file server and its client-side API without a cache mechanism. The second stage is to design and implement a cache mechanism.

3. Detailed Specification

3.1 Remote file server

File Path. A user is supposed to specify the file name in the form *IP-address:port/filepath*. The *IP-address* refers to the host where your server is running. *port* is the port at which your server is listening, which can be any number between 10000 and 10100. *For security reasons, you should only allow the filepath to include a single file name without a directory path, i.e. check that the filepath does not contain "/".* This means that the server assumes that the file to be accessed is located in the same directory as the server program. This is important since, in this way, any possible disaster caused by your program is restricted to that specific directory.

Operations. The server will implement *read, write, lookup,* and *getattribute* operations. Specifically, the interface which the server will export to your client-side library should be:

- **res = read(filename, offset, n)**, where result *res* contains (1) a sequence of bytes read (if the end of the file is encountered, up to there) and (2) the last modified time of the file.
- **res = write(filename, offset, data),** where result *res* contains (1) a boolean which tells the result of operation (i.e., success or failure), and (2) the last modified time of the file.
- **res = lookup(url)**, where result *res* contains (1) a boolean which tells the result of the operation (i.e., file exist or not exist), (2) the size of the file, and (3) the last modified time of the file. The argument `url` has form *IP-address:port/filename*.
- **res = getattribute(filename)**, where result *res* contains the last modified time of the file.

Note these are *not* Java methods. They are protocols exchanged between your server and your client-side library. For example, to perform the read operation, the client will send a request message to server, which carries filename, offset, and n. The server will perform the read operation and return a reply message to client, which carries the sequence of bytes read and the last modified time of the file.

Statelessness. The server should be *stateless*. As far as this requirement is satisfied, any reasonable design is permitted. The design of the server may not differ between the two stages, except that new operations may be added, such as those for specifying the mode (read or write) with which a file is being used.

3.2 Client-side API

3.2.1 Stage 1

Operations. The realization of a client-side API which interacts with a stateless server is the main topic of this project. It is required that you realize the following interface as Java methods.

- **fh = open(url)**, where *fh* is an integer file handle. The argument `url` has form *IP-address:port/filename*.

- **res = write(fh, data)**, where *res* contains a boolean which tells the result of the operation (success or failure).
- **res = read(fh, byte [] data)**, where the resulting data is read to the byte array *data* (if the length of the array *data* is *n*, then *n* bytes are to be read). *res* indicates the number of bytes actually read.
- **res = close(fh)**, where *res* contains a boolean which tells the result of the operation (success or failure).
- **res = isEOF(fh)**, where *res* contains a boolean which tells if the end-of-file is reached.

The Java interface for the client-side API (which you will implement) is given in the file fileSystemAPI.java. The file fileSystem.java provides a simple example implementation of the client API of the file system. It provides the operations *open*, *read* etc. on a *local* file. These must be replaced by operations that communicate by means of TCP with the file server.

The following gives the required semantics.

- **Open.** When you open a file, you obtain a file handle that uniquely identifies the file. At the time of *open*, you may specify the mode as reading or writing - this is not necessary in Stage 1.
- **File pointer**. Once you open the file, you keep a pointer. The pointer moves on as you read/write data, starting from the beginning and finishing at the end of the file.
- **Close**. You should discard the file handle and remove any client state (e.g., file pointer).

Fundamental Requirement. Any client which uses the interface above should be able to read/write a file remotely over the Internet.

3.2.2 Stage 2

Operations. These are the same as Stage 1, though you may add a read/write mode in the *open* operation for the control of caching.

Semantics. Here you should add a client-side cache mechanism to your original program. Specific requirements are as follows.

- **Open a file for read**: First check if the file is cached. If not, read operations are sent to file server and the results of the read operations are cached. If the file is cached (i.e., the file has been opened previously), client-side contacts the server to obtain the last modified time of the file. If the last modified time is later than that of the cached file (i.e., the cached file is stale), the cached file is discarded, subsequent read operations are sent to the server, and results of reads are cached. If the cached file is not stale, client-side will read from the cache.
- **Open a file for write**: Write operations should write to local cache. When the file is closed, the cached data must be flushed back to the file server.

4. Java Files for the Project

The Java interface for the client-side API for the file system is given in the file fileSystemAPI.java. The definition of the class *filehandle* is given in the file filehandle.java. The program in the file checkfh.java can be used to check whether the filehandle class works correctly.

The file fileSystem.java provides a simple example implementation of the client API of the file system. It provides the operations *open, read* etc. on a local file. These must be replaced by operations that communicate by means of TCP with the file server.

The program FileSytemTest.java is for testing your client API. It tests read and write operations by reading and writing to the file testfile.txt.

The program ReadTest.java is for measuring the time taken to read a large data file. The program repeatedly calls the *read* operation in the client API to read the file largedata.txt from the file server. It includes timing statements allowing you to measure the performance of reading the file.

5. Deliverables

- **Stage 1 program (80 points)**
- **Stage 2 program (40 points)**
- **Documentation (15 points):** You should add plenty of comments to describe the purpose of your code. Comments should be added above every class, above every method in a class, and in line where it is not obvious what you are trying to do.
- **Report (15 points):** You report should include the following:
    - Describe how your software can be run. If you hard-wire IP address, port number, etc., that is OK but write a note on this.
    - Provide measurement results of reading largedata.txt for Stage 1 (without cache) and Stage 2 (with cache).
    - Give a brief analysis of the result of your measurements, including discussions on how your cache mechanism contributes to the difference in performance.

6. Project Submission

Your programs must compile and run on pyrite.cs.iastate.edu.

You will submit your Java files and the report. Put all your Java files (including the test programs) and your report in a folder. Then use command
`zip -r <your ISU Net-ID> <project_folder>` to create a .zip file. For example, if your Net-ID is `ksmith` and `project2` is the name of the folder that contains your Java files and report, then you will type
`zip -r ksmith project2` to create file `ksmith.zip`, which you will submit on canvas.