15-440 Spring 2023
Project 2: File-Caching Proxy

Important Dates:

Project Handout:	Tuesday February 7, 2023
Checkpoint 1 due:	Tuesday February 14, 2023, 11:59 PM EST
Checkpoint 2 due:	Tuesday February 21, 2023, 11:59 PM EST
Final Due:	Thursday March 2, 2023, 11:59 PM EST
Submission Limits:	15 Autolab submissions per checkpoint without penalty (5 additional with increasing penalty)

Important Guidance

This project will need to be done in Java 8, in a 64-bit Linux environment, e.g., Andrew Unix servers. *It will not run on Windows or Mac !!!*

Keep your AFS space secured. You will design and implement a server that lets a remote client read, modify, and delete files in your file space with no security provisions! Please be careful, and do not leave your servers running.

This project is about design. There are many things unspecified. You will have to make wise choices that balance implementation complexity and performance, while meeting all constraints that are specified.

What you will learn from this project

You will learn how to:

- Design a caching protocol
  - Based on one of the strategies covered in lecture (e.g., check on use, callbacks, etc.)
  - Which robustly handles multiple concurrent clients
  - Which ensures open-close session semantics on concurrent file access
  - Which uses LRU to manage a fixed size cache containing variable sized files
- Design and implement a distributed system
  - For whole file caching
  - Using Java RMI, Java threading, and concurrency management techniques
  - Which emulates C file operations on locally-cached files

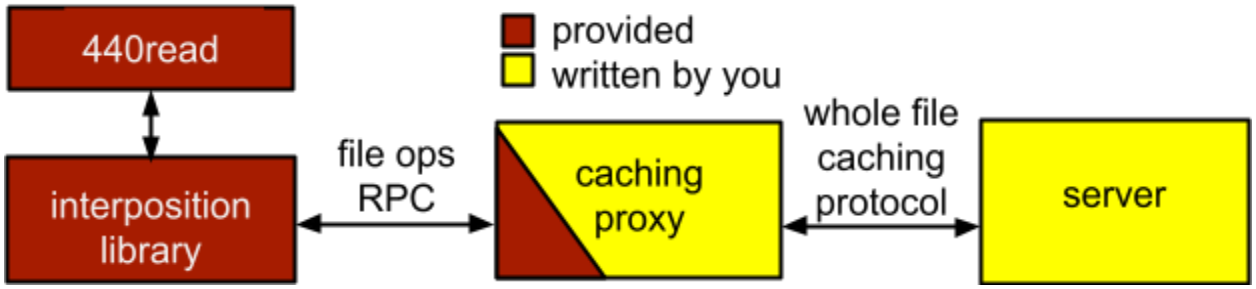
Introduction

Caching is a great technique for improving the performance of a distributed system. It can help reduce data transfers, and improve the latency of operations. This project extends the simple remote file operations system we built in Project 1 to now perform caching as well. This project will continue to use existing binary tools (e.g., `440read`) and interpose on their C library calls.

We will provide a working interposition library. However, instead of connecting directly to a server, this library will connect to a caching proxy, which you will write. This proxy will in turn connect to the server, which you will write as well.

The proxy will handle the RPC file operations from the client interposition library. It will fetch whole files from the server, and cache them locally. You need to define the protocol used between the proxy and the server, and how the cache will be managed. Remember, this protocol should offer open-close session semantics on whole files. Therefore, the protocol should not be implemented at the level of individual `read`, `write`, and `seek` operations.

In addition to the client interposition library, we will provide a Java class to handle the low-level RPC serialization with the client library. It will call your code to actually perform the operations. You will write a class that will implement an interface providing methods for `open`, `read`, `write`, ... operations that behave similar to the C functions. Your proxy code will be responsible for fetching and caching needed files from the server.



Requirements/Deliverables

We will provide:

- We provide several components for this project. These require that the project be implemented in Java, in a 64-bit Linux environment, e.g., Andrew Unix servers
- We will provide you a set of binary tools that perform file I/O using C library calls
- We will provide a client library to interpose on C library file operations, and transform them into RPCs to the proxy
- We will provide you a Java class (`RPCreceiver`) that implements the stubs for the RPCs from the client library. This class will perform data serialization and deserialization, transform data into a more Java-friendly form, and call methods you provide to actually execute the operations.

You will create:

- You will create a server that stores and provides access to a set of files. Your server should operate at the granularity of files (not individual operations), and handle concurrent connections from multiple proxies.
- You will create a proxy that sits between the client and the server. It should use the `RPCreceiver` class we provide to communicate with the clients. It will handle the file

operation RPCs from one or more concurrent clients, fetch needed files from the server, and push modifications back to the server. It will provide C file operation semantics (`open`, `read`, `write`, …, use of file descriptors, maintain current position in file, etc.) to the clients, but talk to the server at a whole-file granularity. It will also perform caching to reduce latency and total data transfers.

Your code should do the following:

- Your proxy needs to emulate the following C library calls (with some simplifications and modifications, see below): `open, close, read, write, lseek, unlink`
- You are free to design your own protocol between proxy and server. However, you must implement whole file caching, some method of ensuring caches are not stale, and LRU eviction/replacement policy. Document your design choices.
- Your code should allow concurrent read and write access to the same file by different clients, but should ensure that while a client has a file open, it will see a stable version (i.e., it will not be affected by modifications or deletions caused by other clients).
- The semantics of your system should not change depending on whether concurrent clients are connected to the same proxy or to different ones.
- You are required to use Java RMI for the RPCs between your proxy and server.
- Your server will be provided two command line arguments: <port> and <rootdir>. <port> will be the port your server should use to listen for proxy connections (i.e., create a Java RMI registry on this port). <rootdir> points to a directory tree populated with the initial files and subdirectories to serve. Files outside this directory tree should not be served or be visible to proxies and clients. Note that your server is free to use any internal representation of these files, and is not required to update the files in <rootdir> as clients make changes. You may update them if you wish, but are free to track changes in other ways.
- Your proxy will be provided four command line arguments: <serverip> <port> <cachedir> and <cachesize>. <serverip> and <port> are the server's IP address and the port it uses (i.e., the Java RMI registry's port). <cachedir> is a directory your proxy must use to store its cache data. The actual files and formats are totally up to you, but the total contents should not exceed <cachesize>, the size of the cache in bytes.
- Your proxy should implement an LRU cache replacement policy.

Submission and grading:

- You will be graded primarily on correctness of the file operations and caching. Performance is secondary, but will also be tested.
- This project will use an autograder to test your code. See below on how to submit.
- You need to submit a short (1-2 pages) document detailing your design. See below.
- The late policy will be as specified on the course website, and will apply to the checkpoints as well as the final submission
- Coding style should follow the guidelines specified on the course website

Checkpoint 1 (20%) **Due: 11:59 PM, Tuesday February 14, 2023**

Checkpoint 1 requires that the client-facing aspects of your proxy are working properly. In particular, all of the file operations need to work properly, providing expected outputs and error codes. Just for this checkpoint, instead of fetching files from the server, the proxy will serve the files in its working directory (imagine this is a prepopulated / warmed cache). Thus, you will not implement a server for this checkpoint, and the proxy does not need any command line arguments. You will not need to worry about cache size limits. However, your proxy must be able to handle concurrent clients.

Checkpoint 2 (20%) **Due: 11:59 PM, Tuesday February 21, 2023**

Checkpoint 2 requires you to implement the server and design your caching protocol. Your system should be able to perform basic read caching of whole files and be able to push modifications to the server. However, interactions between reads and writes, and invalidation of cache entries will not be tested. Your proxy should accept the command line parameters specified in the requirements, but cache size limits will not need to be implemented for this checkpoint. Your server should be able to handle multiple concurrent proxies.

Final (60%) **Due: 11:59 PM, Thursday March 2, 2023**

Full implementation of all of the required features. All of the file operations need to work properly in your caching system. Your proxy should ensure freshness of the cache, and implement an LRU replacement policy to keep total cached state within the limits specified on the command line. Performance of your system will be tested across various high latency and low bandwidth scenarios. (40%)

You will also need to write and submit a 1-2 page document, describing the major design aspects of your project, including the protocol between proxy and server, consistency model as seen by the clients, how you implemented LRU replacement, techniques used to ensure cache freshness, and how these affect your system's performance. Highlight any other design decisions you would like us to be aware of. Please include this as a PDF in your final submission tarball. (10%)

Your final source code will also be graded on clarity and style. (10%)

Submission Process and Autograding

We will be using the Autolab system to evaluate your code. Please adhere to the following guidelines to make sure your code is compatible with the autograding system.

First, untar the provided project 2 handout into a private directory not readable by anyone else (e.g., ~/private in your AFS space):

```
cd ~/private; tar xvzf ~/15440-p2.tgz
```

This will create a 15440-p2 folder with needed libraries, classes, and test tools. You should create your working directory in the 15440-p2 directory. It is important that from your working directory, the provided library and java classes should be available at ../lib.

Write your code and Makefile in your working directory. You must use a makefile to build your project. See the included sample code for an example. You will need to add the absolute path of your working directory and the absolute path of the lib directory to the CLASSPATH environment variable, e.g., from your working directory:

```
export CLASSPATH=$PWD:$PWD/../lib
```

Ensure that by simply running "make" in your working directory, both your proxy and server classes are built. Please name the class implementing your proxy “Proxy" and the class for your server "Server". Make sure the .java and generated .class files are in your working directory (i.e., not in a subdirectory). Both of these classes should implement main. Do not place your classes in a java package! Leave them in the default package. This naming convention and relative file locations are critical for the grading system to build and run your programs.

To hand in your code, **from your working directory,** create a gzipped tar file that contains your make file and sources. E.g.,

```
tar cvzf ../mysolution.tgz Makefile Proxy.java Server.java
```

Of course, replace these with your actual files, and add everything you need to compile your code. If you use subdirectories and/or multiple sources, add these. Do not add any files generated during compilation (e.g. the .class files) -- just the clean sources. Also, do not add the class files, .so file, or binary tools that we have provided -- these will be installed automatically when grading. To work correctly with Autolab, when extracted, your tarball should put the Makefile and sources in the current working directory.

You can then log in to **https://autolab.andrew.cmu.edu** using your Andrew credentials. Submit your tarball (`mysolution.tgz` in the example above) to the autolab site. Note that each checkpoint shows up as a separate assessment on the Autolab course page. For your final submission, include your write up as a PDF file in your tarball.

How to Use the Supplied Libraries and Classes

We provide a set of binary tools that make use of low level file operations using the standard C library. These are a subset of the tools we used in Project 1. You should be able to run these on the local file system -- under the same conditions, using your caching remote file system should provide the same results. You can (and are encouraged to) create additional test programs to really test the corner cases to make sure you will pass the hardest tests we can think of.

To run these tools on the remote file system, we interpose on the C library calls using `LD_PRELOAD`. We have provided the interposition library, `lib440lib.so`; this is used just like `mylib.so` in Project 1:

```
LD_PRELOAD=../lib/lib440lib.so ../tools/440read foo
```

The `lib440lib.so` will connect to the proxy. You should set the environment variable `proxyport15440` to indicate which port to use. We have provided a Java class, `RPCreceiver`, that implements the proxy's side of the client RPCs. It, too, expects the `proxyport15440` environment variable to be set. Both use a shared secret pin to authenticate connections. Set the `pin15440` environment variable to a secret 9-digit pin of your choice on both the server and the client.

Arguments to your proxy will be provided like in this example:

```
java Proxy 127.0.0.1 11122 /tmp/cache 100000
```

with a server address of 127.0.0.1, port 11122, cache directory /tmp/cache, and 10^5 byte cache size limit.

Server arguments will be provided like in this example:

```
java Server 11122 fileroot
```

with a server port of 11122 and serving files in fileroot.

You will write several classes to implement your proxy. You should provide a class named Proxy that implements a `main()` method. This should take 4 command line arguments as specified in the requirements. To use the `RPCreceiver` class, instantiate a new `RPCreceiver`, and call `run()`. Since `RPCreceiver` implements `Runnable`, you can also start it as a separate thread. `RPCreceiver` itself will listen on the proxy port, and launch additional threads to handle clients.

You need to create a class that implements the `FileHandling` interface we have defined. This is a set of Java methods that correspond to the file operations you need to support. `RPCreceiver` will call your class methods to actually do the work for the RPCs. One additional method, `clientdone()` is used to tell your code that the client has gone away (so you can clean up any state).

`RPCreceiver` will need an instance of your `FileHandling` class for each client. You need to provide a "factory" class to the constructor of `RPCreceiver`. The sole purpose of your factory class, which must implement the `FileHandlingMaking` interface, is to return new instances of your `FileHandling` class. For each client that connects, `RPCreceiver` will call `newclient()` on your factory to get a new instance of your `FileHandling` class, and then spawn a new thread to handle the client. All RPC operations from a particular client will invoke the `FileHandling` methods on the associated class instance. Your code needs to be thread safe and handle multiple clients concurrently.

For your server, you need to create a class called `Server` that implements `main()`. It should take two command line parameters, a port and the root directory of the initial set of files to serve. You are free to define the protocol between your proxy and server, but it should be at the level of whole files, not individual operations, and make use of Java RMI for RPCs. You should

instantiate a Java RMI registry that listens on the specified port. Your server must be able to handle multiple proxies concurrently. Your server will not use any of the supplied libraries or classes.

`FileHandling` interface

The `FileHandling` interface we provide describes the set of Java methods you will need to implement to service the client file operation RPCs. The six main methods correspond directly to the C library file operations `open, close,` etc., with some minor modifications. First, the interface is simplified to use Java constructs. Instead of null-terminated character arrays, all paths are specified as Strings. For operations requiring a (`void*`) buffer and a length, the corresponding Java method will take a byte array as an in/out parameter.

Secondly, the options for `open` have been simplified. Instead of the set of option flags (some combinations of which are not meaningful), the open method will take only one of four options: `READ` (read-only), `WRITE` (read/write), `CREATE` (read/write, create if needed), and `CREATE_NEW` (read/write, but file must not already exist). These are defined in the `OpenOption` enum. Furthermore, the `mode_t` bits are not used for this project.

Instead of a separate return value and error code, all of the operations should return a non-negative value on success and in case of error, a negative value that indicates the error. For your convenience, the `FileHandling.Errors` class defines some constants corresponding to error conditions you may need to report. Note that these are just negative versions of the error codes reported in `errno` for the C operations, so feel free to use any other standard error code you feel is appropriate (the negative versions, of course).

Ensure your implementation adheres to the description of the operations in the man pages, except for the modifications noted above. Remember that `read` and `write` should return the actual number of bytes read/written on success, while `lseek` returns the new position in the file.

Finally, the `clientdone` method is not an actual file operation, but is a way for `RPCreceiver` to let your class know when a client has left, so you can clean up any state.

Notes / Hints

- Make sure your design adheres to the requirements specified here. Thinking through all of the corner cases, designing your system carefully to meet all of the requirements, and

testing on your own are the only way to ensure full marks. Do not just try to pass the autolab test results. E.g., Think carefully about what LRU means. What does file use mean? Does it make sense to keep a file cached that no one can possibly ever access again?

- Do not hardcode the server IP address / port in your proxy or server. Instead, use the IP address and port specified as command line parameters (see requirements). Remember to set the environment variable `proxyport15440` so `lib440lib.so` can find your proxy, and the `RPCreceiver` class can listen on the right port.
- Note that you may have many clients connected to a single proxy, and multiple proxies connected to the server. Make sure you can handle concurrent access to the same files, both for read and write, following open-close session semantics. You have some design freedom in how exactly you handle this, but you need to state and justify your choices.
- Remember that with whole file caching, once a reader opens a file, it should see the same consistent view of the file, even if it is deleted or changed elsewhere. However, the updated version should be visible to clients that open the file subsequently. For example, if client A has the file open while client B writes to the file, then A must continue to see the old version. However, another client C, which opens the file after B writes and closes it, must see B's modifications.
- You have some flexibility in how you want to handle concurrent writes to the same file. Hint: AFS-style concurrency or write locks / leases are acceptable, as long as readers are never blocked.
- Remember to test cases where concurrent clients connect to the *same* proxy and where they connect to *different* proxies. Ensure that the behavior remains exactly the same.
- Make sure that (a) caching works, and (b) your protocol between proxy and server is efficient. In other words, be careful not to defeat the benefits of caching by having a very chatty protocol that increases latency!
- Please note that Java has many different classes/APIs for accessing files, that vary greatly in terms of ease of use and richness of features. What you may have used in previous classes to easily read a file may not be the best choice for this project. You may want to consider `RandomAccessFile`, which provides operations close to (but not exactly like) those used in C. Please take a look at online documentation on `File` and `RandomAccessFile` for more information.
- On a related note, semantics of Java file operations differ from those of C operations. In particular the errors signaled and the return values may be different. *Be careful that your code provides return values and errors according to C file semantics, as expected by the C client programs.* Hint 1: How does Java signal that no bytes were read? Hint 2: What happens when you open a directory in C?