CS 677 Distributed Operating Systems

Spring 2014

Programming Assignment 3: Asterix and the Olympic Games (Fault Tolerant edition)

Due: 5pm, Wed April 30, 2014

- You may work in groups of two for this lab assignment.
- This project has two purposes: to familiarize you with concepts in caching and fault tolerance
- You can be creative with this project. You are free to use any programming languages (C, C++, Java, python, etc) and any abstractions such as sockets, RPCs, RMIs, threads, events, etc. that might be needed. You can also build on the code that you wrote for the previous lab. You have considerable flexibility to make appropriate design decisions and implement them in your program.

A: The problem

- After the tremendous success of the winter Olympic games, the Gauls are preparing for the summer Olympic games. The summer games are expected to be singificantly more popular and draw more traffic than the winter games.

Part 1: Caching and Cache Consistency

The Gauls are continue their effort to prepare for the expected surge in the number of smart stones accessing sports information. Both performance and reliability enhancements are planned. Previously the front end stone server was replicated but each request still needed to make access to a backend database which resulted in slow disk I/Os To enhance performance, Obelix has decided to add a cache to each front end server. The cache stores a copy of recently accessed scores for each sport. When a tablet makes a request to the stone server, the front-end first check the cache to see if the requested data is cached. In the event of a cache hit, the cached data is used to service the request. Since the cache is a memory cache, cache hits avoid the need to access the database server and enhance performance. However, if the request results in a cache miss, the front-end servers must request data from the backend server like before. Like before Cacofonix sends sports score updates to the stone server. Since the scores are frequently updated, cache consistency is a must. The Gauls do not like their fish or their sports scores to be stale! To serve the scores fresh, Obelix has decided to implement two approaches to cache consistency: push-based consistency and pull-based consistency. In push-based consistency, the master front-end server tracks what scores are cached within each cache and upon receiving an update, sends a cache invalidate to any cache that caches that data. Cache invalidate messages cause the cache to remove the corresponding item from the cache and a subsequent request causes a cache miss and the item is brought back into the cache. In pull-based consistency, it is responsibility of the cache to periodically poll the front-end server to cehck if the data has changed. If data is unchanged, it remains in the cache, otherwise the stale data is removed. The frequency at which the caches poll the server determines the degree of freshness. You are free to implement any approach to determine the poll

frequency---the frequency can be fixed or determined based on popularity (more popular sports see more frequent polls).

Your system needs to support both push and pull-based techniques. However when the system starts up, you need to provide a configuration option that specifies which of the two should be used.

Part 2: Fault Tolerance

To ensure reliability, the front-end servers must be able to handle failures. In this lab, only crash failures of the front end need to be handled. Assume that one of the front end servers can fail at any point. Implement heartbeat messages to detect the presence of a failure and have the remaining front end take over the tasks of the failed server. This may involve (i) informing the Cacofonix server to send subsequent updates to the new server (if needed), (ii) informing clients to send subequent requests to the new server (you can assume that clients register themselves with the Stone server at startup and hence the servers know all clients accessing the server). Your code must be able to handle the crash failure of either front-end server. Further, your code must be able to handle recovery -- when the crashed server comes back up, it should be able to resume operation by taking over its original responsibilities (e.g., roughly half the clients are reassigned to the server and if the server was receiving updates previously, it should do so again). Like before, assume that the front end servers not only receive requests for scores but one of them also receives score updates from the Cacofonix process and then sends an update to the database tier. Further, while the API exposed by front-end servers is identical to labs 1 and 2. You may make suitable modifications to the interface for tasks related to client registrations, caching and fault tolerance. as well as design any internal interface between the front-end and back-end processes for this purpose (you can deisgn it any way you wish and this interface should be documented in your README file).

Like before assume that there are *N* tablets, each of which is a client, that needs to be periodically updated with sports scores. (*N* should be configurable in your system).

Like before, Cacofonix, the village bard, is responsible for providing Obelix's server live updates from the olympic stadium, which he does by singing the scores and thereby sending updated scores to one of the front-end servers.

Optional Extra Credit:

Assume four front-end servers and a single database server. Assume that the village is infiltrated with a Roman spy who may corrupt one of the servers in an abritrary manner. Implement the Byzantine generals problem discussed in class for the four servers to handle byzantine faults on any one server. This part is optional and is sigifnciant more challenging than simply handling crash faults. If you decide to implement this part for extra credit, plan your time carefully.

Requirements:

1. You need to implement BOTH Part 1 and Part 2.
2. All the requirements of Project 2 still apply to Project 1, except that you no longer needs vector clocks or clock syncronization.

- **Other requirements:**

  No GUIs are required. Simple command line interfaces and textual output of scores and medal tallies are fine.

You are free to develop your solution on any platform, but please ensure that your programs compile and run on the edlab machines (See note below).

B. Evaluation and Measurement

1. Compute the average response time of your new server as before. Comment on the performance improvements, if any, observed due to caching.
2. Design tests to compare pull and push-based consistency. Make observations on much stale data is observed in pull-based consistency in your design.
3.
4. Design test to show the reliability of your system when a front-end server fails. Demonstrate that recovery can also be performed.

5. Make necessary plots to support your conclusions.

• C. What you will submit

• When you have finished implementing the complete assignment as described above, you will submit your solution in the form of a zip file that you will upload into moodle.
• Each program must work correctly and be **documented**. The zip file you upload to moodle should contain:

1. An electronic copy of the output generated by running your program. Print informative messages when a client or server receives and sends key messages and the scores/medal tallies.
2. A seperate document of approximately two pages describing the overall program design, a description of "how it works", and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and sketch how they might be made). You also need to describe clearly how we can run your program - if we can't run it, we can't verify that it works.
3. A program listing containing in-line documentation.
4. A seperate description of the tests you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly.
5. Performance results.

• D. Grading policy for all programming assignments

1. Program Listing
   works correctly ---------- 50%
   in-line documentation -------- 15%
2. Design Document
   quality of design and creativity ---------- 15%
   understandability of doc ------- 10%
3. Thoroughness of test cases ---------- 10%
4. Grades for late programs will be lowered 12 points per day late.

• Note about edlab machines

• We expect that most of you will work on this lab on your own machine or a machine to which you have

access. However we will grade your submission by running it on the EdLab machines, so please keep the following instructions in mind.

- You will soon be given accounts on the EdLab. Read more about edlab and how to access it here
- Although it is not required that you develop your code on the edlab machines, we will run and test your solutions on the edlab machines. Testing your code on the edlab machines is a good way to ensure that we can run and grade your code. Remember, if we can't run it, we can't grade it.
- There are no visiting hours for the edlab. You should all have remote access to the edlab machines. Please make sure you are able to log into and access your edlab accounts.
- IMPORTANT - No submissions are to be made on edlab. Submit your solutions only via moodle.

Stumped?

1. Who are the Gauls? Read about them on Wikipedia.
2. Stumped on how to proceed? Review the comic book Asterix at the Olympic Games from your local library. Better yet, ask the TA or the instructor by posting a question on the Piazza 677 questions. General clarifications are best posted on Piazza. Questions of a personal nature regarding this lab should be asked in person or via email.