Operating Systems and Distributed Systems

Fall 2022

Project 2: Distributed Image Server

The purpose of this project is to get you started on distributed system programming. You will extend the program you developed in the previous project into a distributed setting. Consider a photo server that stores lots of images of the original size. The client can download one or more images from the server, specifying the desired size. The server will resize the image(s) before sending them to the client.

We will provide 20 servers for the entire class to perform the experiments. All original images (about 50,000 files) are copied to all servers under /ImageNet($IMG\_PATH). There is a hierarchical path structure. You can login and explore the path structure under the path. Each image is addressed by a key of the format abcdef, corresponding to a file path of $IMG\_PATH/ab/cd/ef.JPEG. You need to process 2,000 images according to the key and resize parameters in resize.csv we provide.

The client program, running anywhere that can access the image server, (e.g. same server, another server, or your laptop), can make an RPC call to request data from the server. To get a single image the client calls the server with (key, width, height) as parameter, and the server returns the resized image as a byte array.

You will need to complete an *image resize and download* task under different settings based on the RPC framework. We highly recommend using GO for this project, but feel free to use C++ instead.

**Part 1** **Downloading images, one at a time, from a single server**

You have a list of images your want to download contained in resize.csv in the csv format, e.g. abcdef,128,128

You want to download all files listed in the file, from a single server. Note that in this part, the server interface, GetSingleImg, only servers a single image at a time. Notice that the server should be able to handle multiple clients at the same time, using a thread pool.

Implement a basic client-server system based on the RPC framework to complete an image resizing task. As a single RPC call,

* client sends a request of a single image from a single server
* server accepts and processes the client's request and returns the result to client after processing. You need to think about where to put your task queue? Where to use server threads?
* The client saves the result returned by server on its local disk in the current directory with file name “{key}.jpg”.

If the server does not respond for any reason, the client can try other servers, but it uses a single server at a time. The following procedure will help you develop this program.

**Todo 1: Getting the Single client – Single server work**

You should first implement a basic single-client-single-server system described above. Making sure a single client thread can sequentially make multiple requests correctly.

**Todo 2: Mult-thread client**

Then you extend your results by making the previous client program multi-threaded. Use a flag to control how many threads you will run for your client program. Make sure that the clients write the files correctly.

**Todo 3: Multiple client processes**

Use a main() function that fork()’s multiple processes, each containing multiple threads. Note that as these processes do not share memory, you need to find a way to coordinate them so they are downloading different files in the list.

**Todo 4: Performance evaluation**

Based on your previous implementation, you need to evaluate the performance and factors affecting performance. For example, number of clients and number of requests of each client may affect transmission performance such as latency and throughput. Also, there may be a large variation on each call, and there might be failures.

Please summarize and analyze the reasons behind the behaviors you find interesting. However, your report should at least contain: throughput vs. number of client / server threads; number of client / server threads vs. latency (mean and variations); multiple processes vs. multiple threads, as well as the server queue lengths over time. Of course, you should be clear about your settings (e.g. where do you run your client / servers? What is the network bandwidth / latency? What kind of client do you use?)

**Todo 5:** **Automate the above experiments**

Like in Project 1, you should try to automate everything using Makefile. Specifically, write a deploy\_server target to copy your server executable to the server(s) you want to run, start/stop\_server targets to start the server process (for the above two targets, think about the object file output to make Makefile work as expected); as well as get\_server\_stats to download the server output data you need (e.g. the queue length you will need to plot your figures). You will want to learn to use ssh and configure the public-key based password-free login to make these make targets work.

With this make file, the distributed experiments will become easier. Using these targets you can fully automate the Make process.

Like in the previous part, you should write a Makefile target make part1\_pdf to generate the pdf version of report with all write-ups and plots. The report should not exceed 2 pages, and you should explain your observations, as well as why you think it behaves that way.

**Part 2 Optimizing performance**

There are some obvious performance limitations in the solution in Part 1. E.g. each time you can download a single image only; you can access only a single server, while there are 20 available, etc.

In this part, you will improve the image resize task you implemented in part 1 to optimize for performance. One client program is allowed to use all servers, and you are allowed to add a new RPC interface GetMultiImgs. This interface can take a list of (key, width, height) specs, and returns all resized images in the list as return value.

**Todo 1: Core coding**

1. Add the GetMultiImgs interface and implementation to the server program in Part 1. You may want to use streaming RPC (aka. Asynchronized RPC) to improve performance as you can return values in small pieces to the client. You can look up the Go tutorial for sample code on streaming RPC.
2. Implement the matching client.

**Todo 2: Performance evaluation**

Repeat the evaluation in part 1, and observe the performance improvements. You should add factors specific to part 2, such as concurrent servers, batch sizes, asynchronous RPC vs. synchronous, etc. in the evaluation.

**Todo 3: Automation and reporting**

Same as Part 1, you should automate the report generation process as much as possible.

**Submission and Grading:**

We have 20 servers for the entire class and one account for each group. Each group will choose a leader to get your account and password from the TAs (ssauterne@qq.com or wwdkl20@outlook.com).

Each server has:

0. DATA partition for image dataset.

1. Code partition for student code, etc.

This code partition is not on these 20 servers, but will be mounted automatically when students log in to the servers. 20 servers share this partition.

3. public partition: a local partition on the server, which can be used to optimize the data set to increase the read/write speed, and is shared by all groups.

Students have one account per group.

0. Each account can access the 20 servers

1. Each account can access the Code partition. [read-write permission]

The location is: /home/username

2. Each account can access the DATA partition on each server.[read-only]

Location: /ImageNet

3. Each account can access the local public partition on each server [read-write permission].

Location: /osdata

[Note]

Do not reboot the machine without special circumstances.

Like Project 1, you should put everything into a Tsinghua gitlab repo and give TAs access. You should submit your repo URL, as well as the pdf version of your report to learn.tsinghua. Again, each group should only submit EXACTLY ONCE on learn.tsinghua.

You will be graded by the correctness, clarity of explanation and experiment data in your reports, as well as the overall performance of your program. Specifically, for each part:

1. Correctly generate and download all image files worth 30% (15% each part).
2. Report worth 60% including
   1. All performance figures and analysis worth 15% each Part.
   2. Writeups of observations and reasons why it happens, worth 15% each part.
3. Overall performance and code quality etc. worth 10% (subject to the TAs judgement).