Assignment 2: Latency-consistency tradeoffs

Updates

Thu Mar. 7:

1. Corrected name of image in second backend instance.
2. Added note that frontend code is open but backend code is closed.

Overview

In this assignment, you will explore the relations between consistency, latency, and quorum size. You will compare the performance of three quorum sizes on the client side against stores with different consistency guarantees and different latency profiles.

At its heart, this assignment is a puzzle. The backends are black boxes—you can only observe their replies and latencies. The frontend offers several quorum sizes for querying the databases. By considering how the results vary with different frontends, you can infer what is occuring in the databases.

I recommend doing this in at least two sessions

This assignment has several parts, the first a bit tedious, the others more analytic. I suggest setting aside time early to do the tedious setup portion first, check that it has succeeded, then set it aside. Come back later for the analytic parts.

Part 1: Setting up

The first step is installing and configuring 13 services. The process is straightforward but it involves a lot of clicks and you have to pay attention at each step, ensuring the correct settings. To help you check your work, I have provided a simple tool to check the configuration once you're done.

Aside: The configuration we will use in this assignment is nonstandard; you would not typically set up a service such that every instance is directly addressable. In fact, good design would typically recommend making instances anonymous, with a load balancer spreading client requests over the instances. A typical OpenShift install does this automatically.

But this assignment requires that the client be able to directly address each instance of the backend, to highlight the effects of the design choices in a reasonable amount of time and with only three instances. These effects are still observable in standard designs but it would take more instances and much more time to show them. We're using a nonstandard design here to make these effects "pop out" with less work.

> **I strongly recommend that you follow my naming conventions exactly. This will simplify running the frontend client later. Do not try to save typing by shortening them.**

What code you can examine (added Thu Mar. 7)

The code for the frontend is open—you are welcome to review it to learn how it works.

The code for the backend is *closed*—the assignment asks you to make claims about the underlying algorithms by analyzing its performance as a black box, not by examining the code.

Setting up a new project

Select `View All Projects` from the dropdown menu in the upper left of the OpenShift console (it will display the name of your current project, likely `My Project`). Use the `Create Project` button (blue, upper right corner) to create a project. Name it `assign2` (this is the top field; you can put whatever you want in the lower, `Display Name` field— `Assignment 2`, for example).

Check that the display name of the new assignment is now your current project (upper left corner). All the following operations must be done in this assignment.

Setting up the first backend instance

You will install 12 backend instances. The first one will require downloading a container image from `hub.docker.com`. The remaining 11 will simply require locally re-instantiating that image under different names.

Deploy the image `tedkirkpatrick/cmpt756-a2-backend:latest`. Name it `a2-backend-a1` and set the environment variable `config` to `A-1`. The cases are significant: The suffix must be lower-case, while the value must be upper-case.

(For a refresher of how to deploy an image, see "Deploy the image" (http://756.cmpt.sfu.ca/756-19-1/ex5-serving-data.html#deploy-the-image) in Exercise 5.)

Check the list of services. You should have exactly one, named `a2-backend-a1`.

Setting up the second backend instance

Now that you've downloaded the container image, you can deploy your downloaded copy directly when you create the remaining replicas.

Select the `Deploy Image` menu item again, but in the dialogue, leave the default `Image Stream Tag` radio button. Enter the following values in the boxes underneath:

Namespace: `assign2`
Image Stream: ~~`a2-backend`~~ `a2-backend-a1`
Tag: `latest`

After entering the tag, the dialogue will expand. Enter the following items:

Name: `a2-backend-a2`

Environment Variable

- **Name:** `config`
- **Value:** `A-2`

Important: Note that the suffix for this instance is now `a2` and the value of the environment variable is now `A-2`.

Press `Deploy` and check that you now have two services, `a2-backend-a1` and `a2-backend-a2`.

Setting up the remaining ten backend instances

Repeat the procedure you just used to deploy ten more instances. Each instance should have a distinct name and value for `config`:

Name	Value of `config`
`a2-backend-a3`	`A-3`
`a2-backend-b1`	`B-1`
`a2-backend-b2`	`B-2`
`a2-backend-b3`	`B-3`
`a2-backend-c1`	`C-1`
`a2-backend-c2`	`C-2`
`a2-backend-c3`	`C-3`
`a2-backend-d1`	`D-1`
`a2-backend-d2`	`D-2`
`a2-backend-d3`	`D-3`

The pattern is simple: For each of the letters, A, B, C, and D, there are three instances, and the name suffix and environment variable uniquely identify each instance using its type and number.

Yeah, it's a lot of clicking. It's essential for the rest of this assignment that the service names and values of `config` match. In the next step, you'll install a tool to check that everything's correct.

Installing the front end

You will be pleased, perhaps even ecstatic, to read that there is just one frontend instance to install.

Although the backend was installed by loading a container image from the Docker Hub registry, the frontend is built from a source repository on GitLab, `https://csil-git1.cs.surrey.sfu.ca/ted/cmpt756-a2-frontend.git`. Use `Python` as the base. Name the application `a2-frontend`.

Refer to the portion of Exercise 3 (ex3-basic-openshift.html) for how to deploy a Python program from source. Hint: Begin with `Add to Project/Browse Catalog`.

Using the frontend to check that the backend instances are correct

The frontend "service" is nonstandard: Instead of a service running on the Internet, we are going to use it as a Linux command-line shell.

Aside: Kubernetes is *very* particular about its pods, requiring any pod to have at least one live Internet port. To satisfy that requirement, the frontend container in fact has a bare Web server running in the background, serving port 8080. We'll never access it but it does exist.

For this step, you only want to use the frontend shell to verify that your dozen backend instances are correctly configured.

To run the shell,

1. Open the `Applications/Pods` page. You will see a list of pods.
2. Locate the pod running the frontend. Its name will look something like `a2-frontend-1-79d6w` and its "Status" will be `Running`. Do not confuse this with pods named `a2-frontend-1-build`, which were used to *build* the frontend and are now marked `Completed`.
3. Click on the pod name. You will see the pod description page.
4. Click on the `Terminal` tab. This will open a Linux shell.

Tip 1: If you leave this page, the shell will close. This can be inconvenient. You might find it useful to open a second OpenShift console window. Use one for controlling OpenShift and reserve the other for running the shell.

Tip 2: If you click in the upper right corner of the terminal window, on the diagonal arrows that appear, you can shift to full screen mode.

For this step, you just want to run the single command,

```
(app-root) sh-4.2$ ./check.py
```

If this command produces messages, you need to reconfigure your backend instances. Either a service is missing, misnamed, or has the wrong environment variable value. Debug the problem and correct it.

If this command produces no output, all your backend instancess are correct. Take a break and reward yourself.

Background: The four database implementations, the three quorum sizes

The system that you just installed simulates four implementations, named A, B, C, and D, of a replicated database. Each implementation is replicated three times, with the instances identified by their numeric suffix, `-1`, `-2`, or `-3`.

These database instances are supposed to be receiving a stream of updates from another, unspecified process. In actual fact, the code is just simulating the effects of that stream using random number generators, but our story is that they're getting updated by other processes.

You will run a client that reads (only) from these databases. The client doesn't do anything useful with the data it gets but simply records two things: (i) the response latencies, and (ii) the response consistencies.

Each implementation is either consistent or inconsistent:

1. **Consistent:** All instances coordinate to ensure consistent results are returned, regardless of which instance the client sends a request to. The "current" value will be the same for all instances.
2. **Inconsistent:** Instances do not coordinate for consistent results. Different instances may sometimes present the updates in a different order. The "current" value may depend upon which instance you send the request to.

In addition, implementations have one of two latency distributions:

1. **Uniform:** Latency is distributed uniformly over a relatively narrow range. There is a maxiumum latency and it is relatively near the minimum.
2. **Ragged:** Latency is distributed exponentially. There is no maximum possible value and extremely large values are possible.

Your goal in this assignment:

> **Use the frontend client to gather latency and consistency statistics from the four implementations and categorize each implementation (A, B, C, and D) as Consistent/Inconsistent and Uniform/Ragged.**

The three quorum sizes

The client supports all possible quorum sizes for a three-way replicated database, using the `--clientalg` option. In every case, the client sends requests to all three instances of the database. The quorum algorithms differ in how many replies each use:

CALL_1
: Take the first response and discard the remaining two.

CALL_2
: Take the first two responses and discard the third.

CALL_3:
: Wait for all three responses.

The consistency measures

The client reports two consistency measures, internal and external.

Internal consistency measures the consistency between responses to a single request. The internal consistency for a CALL_1 request is always 100% because there is only one response and by definition it is consistent with itself. The internal consistency for CALL_2 and CALL_3, is a rough measure of how much the two or three responses (respectively) agree with each other.

External consistency measures the consistency between successive calls to the database. Each database value is timestamped. A sequence of responses where the timestamp is incremented by one for each call is externally consistent. If the timestamp goes backward or jumps by more than one, the sequence is externally inconsistent.

There is no standard way to measure consistency and the exact algorithms I chose are rough. I recommend **that you not focus on the consistency formulas** but instead pay attention to their relationships: **how their values vary by choice of the database and quorum algorithm**.

The frontend gathers the statistics for you. It's up to you to interpret them.

Part 2: Data gathering

Begin by exploring the frontend and its reports in real time.

Exploring the client and the database implementations

Open up the shell in the frontend pod. The frontend client is called `frontend.py`. Try running it now:

```
(app-root) sh-4.2$ ./frontend.py
Calling services
http://a2-backend-a1.assign2.svc:8080/
http://a2-backend-a2.assign2.svc:8080/
http://a2-backend-a3.assign2.svc:8080/
Configuration: A-1
Configuration: A-2
Configuration: A-3

.... 10 latency and consistency values, followed by their mean ...
```

Tip: You can see all the client's options by running it with `-h`. You likely won't use most of them. The two that you absolutely will use are `--clientalg` and `--preset`.

By default, the client contacts database A and uses a quorum size of 3. You specify a different database via the `--preset` option, which takes a database name (in lowercase—sorry about that!):

```
app-root) sh-4.2$ ./frontend.py --preset b
Calling services
http://a2-backend-b1.assign2.svc:8080/
http://a2-backend-b2.assign2.svc:8080/
http://a2-backend-b3.assign2.svc:8080/
Configuration: B-1
Configuration: B-2
Configuration: B-3

... 10 different latencies and consistencies ...
```

Note that the B database was contacted. How do the latencies and consistencies differ from A?

Repeat this for databases C and D.

Vary the quorum sizes

Try a quorum of 2:

```
(app-root) sh-4.2$ ./frontend.py --clientalg CALL_2
Calling services
http://a2-backend-a1.assign2.svc:8080/
http://a2-backend-a2.assign2.svc:8080/
http://a2-backend-a3.assign2.svc:8080/
Configuration: A-1
Configuration: A-2
Configuration: A-3

... 10 more latencies and consistencies ...
```

Vary the combinations of database and quorum size to get a sense of their interactions. Look ahead to Part 3 below to see what questions you will ultimately need to answer.

Gathering systematic data

After exploring the effects informally, gather some systematic results. I have written a script that runs tests on every combination of quorum and database implementation, summarizing each test in a singe line, creating a file `out.txt`:

```
(app-root) sh-4.2$ ./summarize.sh
```

The script uses the `--summary` option of the client, which suppresses all output except the final summary line.

Feel free to review this script and modify it to your interests. As written, it offers good coverage of all the combinations.

The `out.txt` file is terse, designed to be read by an analysis program. You will likely want to read the into your preferred spreadsheet to review the results. To copy the file from Minishift into your host operating system, use the `oc cp` command, after first setting your current project to the `assign2`:

```
$ oc project assign2
Now using project "assign2" on server "https://192.168.99.100:8443".
$ oc cp a2-frontend-1-xx9f7:/opt/app-root/src/out.txt .
tar: Removing leading `/' from member names
```

(Ignore that last message about a “leading `/’, if you happen to get it. This seems to be some side-effect that has no effect on the actual file transfer.)

For a reminder on how to set up `oc`, see “Preparing to run OpenShift” (ex3-basic-openshift.html#preparing-to-run-openshift) from Exercise 3. Replace `a2-frontend-1-xx9f7` with the name of the pod running your frontend.

This sequence will leave the file `out.txt` in the current directory of your host operating system. Import it into your spreadsheet to review the results.

Part 3: Analysis and report

Consider your results in terms of the presentations on consistency, latency, and quorum size in class. Write an analysis that answers the following questions:

1. For each database implementation—A, B, C, and D—what is its consistency guarantee? Which of the two distributions describes its latencies?
2. What is the relation between consistency and latency?
3. What is the relation between quorum size and latency?
4. What is the relation between quorum size and consistency?
5. What is the relation between quorum size, consistency, and the distribution of the latencies?

Describe a "relation" in the above questions by discussing how changing one affects the other and why these changes occur. For example, how does going from an inconsistent implementation to a consistent implementation affect the latencies? *Why* does enforcing consistency in a database change the latency? What about the two algorithms (inconsistent and consistent) causes this effect?

In each case, support your claims with the numbers from your test runs. You don't need a statistical analysis such as a regression. A simple comparison of the averages will be enough.

Your analysis need only be about 4–6 pages long. This length is flexible—if you need a page or two more, use it. But the analysis should not go much beyond this size.

You can structure the analysis in whatever way makes sense to you. You do not have to answer the above questions in order and you can combine answers. You do need to clearly describe the above effects. You are welcome to describe other effects that you find interesting.

Submission

Total points: 20

Length: 4–6 pages.

Number of members: 1 (Individual)

Submit a PDF of your report to CourSys (https://coursys.sfu.ca/2019sp-cmpt-756-g1/+a2-latency)