Lab 0: word count accumulator

28, 2023, 23:59 PDT)

Let's get familiarized with C++ and RPC. This is a simple client-server program using C++ and gRPC.

WARNING! Unlike other labs, this is a solo lab project. You should work on this lab by yourself. It's okay to ask for help when you are stuck too long, but name every student who helped you. If you struggle with this lab, you are not ready for this course.

For any question, please search Piazza first, and if the question hasn't been asked before, please create a public post so that other students can see the interactions and potential answers.

Objective

In this lab, you'll implement a simple distributed application using RPC. This serves as a test of whether you can write C++ applications for distributed systems. You also get to experience a popular RPC framework, gRPC. The rest of the labs will use uncommon research RPC systems.

Some general tips:

Start early. Although the amount of code isn't large, you may get blocked by compilation errors, mysterious bugs, etc. It will take significant time to set up the development and test environment, and getting used to gRPC takes time as well.

Getting ready

Check out gRPC website to learn how to use it. https://grpc.io/docs/languages/cpp/basics/

In this lab, you won't need to install gRPC by yourself.

Install the following packages.

```
sudo apt update
sudo apt install -y \
    git \
    pkg-config \
    build-essential \
    clang \
    cmake \
    libapr1-dev libaprutil1-dev \
    libboost-all-dev \
    libyaml-cpp-dev \
```

```
python3-wheel \
python3-setuptools \
libgoogle-perftools-dev
```

Checking out code

This course uses GitHub Classroom. You should sign up for GitHub and checkout the assignment.

Use the following link to get the assignment from GitHub Classroom.

https://classroom.github.com/a/St7sjl_M

For grading, only the `main` branch will be checked out. Please ensure all your edits are pushed to the `main` branch before the deadline.

WARNING! This lab and your solution should not be shared publicly. It can only be used within this course.

How to build?

Lab0 uses **cmake**. For your convenience, you can just run ***./build.sh*** to create build directory, run cmake, and start build.

The initial build will take a long time to compile gRPC.

After modifying your code, you can just run "***make***" from the ***build/*** directory.

> **If you countered an error saying "c++: fatal error: Killed signal terminated program cc1plus compilation terminated." when running the *./build.sh* script, it might be due to the depletion of memory on your local/virtual machine. There are several options you can take:**
>
> 1. run ***./build.sh*** with a specified concurrent job number by "***-j***" flag. For example, "***./build.sh -j8***" will spawn 8 concurrent jobs for the compilation. You may adjust the job number as you deem suitable. The lower concurrent job number will have a lower memory footprint.
> 2. enable ***swap*** on your local/virtual machine. You might want to check this blog, but we haven't fully tested the instruction listed here.

Introduction

Yo will build two separate programs: client and server.

Client: takes user input as a command line argument

1. **Accumulate**: `./wcClient --dest <serverIP:port> --text <string input>`

   Sends a text to the server. Display the word count for the input text and the cummulative sum of all word counts.

2. **Reset**: `./wcClient --dest <serverIP:port> --reset`

   Reset the word counter used for accumulation in the server.

3. **Shutdown**: `./wcClient --dest <serverIP:port> --shutdown`

   Shut down the gRPC server. This function is already implemented for you.

example:

```
$ ./wcClient --dest 127.0.0.1:12345 --text "Hello! This is a lab0 client."
Word count: 6
Sum of all word counts: 6
$ ./wcClient --dest 127.0.0.1:12345 --text "Exciting! Right?"
Word count: 2
Sum of all word counts: 8
$ ./wcClient --dest 127.0.0.1:12345 --reset
Reset counter.
$ ./wcClient --dest 127.0.0.1:12345 --text "After reset, it should be 6."
Word count: 6
Sum of all word counts: 6
$ ./wcClient --dest 127.0.0.1:12345 --shutdown
Shutdown requested.
```

Server: listens on port and accumulates word count and returns to the client.

```
./wcServer --bind <IP:port to listen>
```

example:

```
$ ./wcServer --bind 0.0.0.0:12345
Server listening on 0.0.0.0:12345
grpcServer shutdown in 3 sec.
grpcServer shutdown.
```

The code

There are two folders: `protos/` and `src/`.

gRPC uses protobuf to define RPC specifications.

- `protos/accumulator.proto`: Protobuf definition for gRPC.

- src/clientMain.cpp: contains main()
- src/rpcClient.h and src/rpcClient.cpp: implements accumulator client

For server binary (wcClient),

- src/serverMain.cpp: contains main()
- src/rpcService.h and src/rpcService.cpp: implements accumulator server

RPCs

In this lab, you should implement four RPCs to support the three functionality mentioned above.

- `AddWordCount(text)`: returns the word count of the text, and the server accumulates the word count.
- `GetAllWordCount()`: returns the accumulated word count.
- `ResetCounter()`: resets the accumulator.
- `Shutdown()`: shutdown the server. THIS IS ALREADY IMPLEMENTED.

Milestones

The full credit for this lab is 5 points.

- Milestone 1 (2 points): correctly working Accumulate function
  - You should implement two RPCs, `AddWordCount` and `GetAllWordCount`, in `src/rpcClient.cpp` and `src/rpcService.cpp.`
- Milestone 2 (2 points): correctly working reset function
  - You should define a new ResetCounter RPC in `protos/accumulator.proto`.
  - Define and implement the RPC handler for this in `src/rpcService.h` and `src/rpcService.cpp`
  - Define and implement the RPC client in `src/rpcClient.h` and `src/rpcClient.cpp`:
  - You can refer to how Shutdown is implemented.
- Milestone 3 (1 point):
  - For the last point, you should extend AddWordCount RPC to return both the word count of the input text and the cumulative word count sum. The goal is to reduce the two RPC invocations to one invocation for **Accumulate** function.
  - You should modify the AddWordCountReply message definition in `protos/accumulator.proto`. Add another int32 field at position 2. This will ensure your client or server can be used with another students'.
  - You should modify `src/clientMain.cpp` to use only 1 RPC call.
  - You may have to modify any files.

save your working wcClient before modifying client codes.

- Hint: Protobuf's message definition can be updated to include new fields while maintaining backward compatibility. (https://protobuf.dev/programming-guides/proto3/#updating)

Make small commits, and you must have a separate commit after each milestone at least. Your submission will be checked by plagiarism check software.