COMPSCI 677 Spring 2022

Lab 1: Toy Store - Design

Team Members: Maoqin Zhu, Yixiang Zhang

Part 1 - Implementation with Socket Connection and Handwritten Thread Pool

1. Communication in Distributed System

In this section, we provide an explanation about how client and server connect with each other remotely.

1.1 Socket Programming

We actually apply a regular socket connection in this project. It still works when multiple clients send request, as each connection from clients will be a separate socket port number on the server side. Sockets are bidirectional connections and the thread can send back the reply on the same socket back to the client.

Specifically, in our design, the “producer”(we will talk about it later) will accept connections and receive requests. It should then send the tuple (request string, socket) to the “consumer thread” via the request queue. After processing, the response will be sent back by the “consumer” thread.

We use the socket module in python to create and use sockets. Please check out the implementation details in our source codes. A few common methods we use are shown below:

1) Server Socket Methods

- s.bind(): binds address (hostname, port) to socket

- s.listen(): sets up and start TCP listener

- s.accept(): passively accepts TCP client connection, waiting until connection arrives (blocking)

2) Client Socket Methods

- s.connect(): actively initiates TCP server connection

3) General Socket Methods

- s.recv(): receives TCP message

- s.send(): transmits TCP message

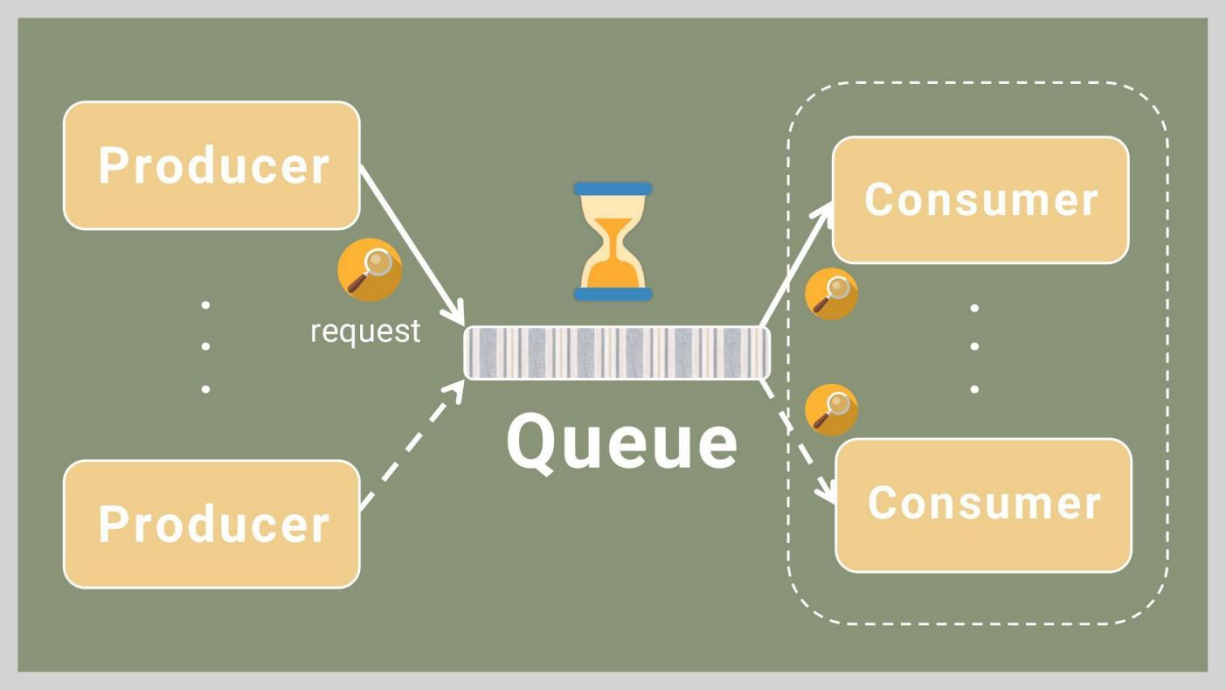
- s.close(): closes socket

2. Handwritten Thread Pool Design

In this section, we provide an explanation about how we create our own thread pool and why it works.

2.1 Define and Create “Producer” & “Consumer”

In this assignment, we could start with solving the Producer-consumer problem, also known as Bounded-buffer problem.



The main idea behind this design is that we consider two processes, which are called the “producer” and the “consumer” respectively. The producer is a master thread, and each time it goes through its cycle and pushes request from clients into the buffer area. The consumer is a thread in the thread pool, and each time it goes through its cycle, requests in the buffer will be fetched and processed by it. We assume the producers and consumers to be connected via a buffer with unbounded capacity.

Specifically, we define two classes in “server.py” to indicate the “producer” and “consumer” respectively. When the server starts up, the master thread will run as “producer”. Then you can type in the size of thread pool, corresponding number of threads will be added into the pool as “consumer”.

2.2 Concurrency & Synchronization

In order to design a concurrent networked server application, we can implement the Producer-consumer framework using both **locks** and **condition variables**. A condition variable is always associated with some kind of lock, which can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

Here we did not import "Queue" from python library, even though "Queue" has encapsulated the behaviour of Condition. That is because our goal is to show you how we use those methods of Condition.

A brief explanation of our code is summarized as follows. If you want to see the details, please check out the source code in "server.py".

Consumer View:

1) Check out if we can acquire the underlying lock using ***if condition.acquire().***

2) Check out if the request queue is empty.

If yes we call ***condition.wait()*** and ***condition.release()***. In this case it releases the underlying lock, and then blocks until it is awakened by a ***notify()*** for the same condition variable in another thread.

3) Producer puts request in queue and calls ***condition.notify()***. Consumer then wakes up, but does not start executing.

4) After producer releases the lock using ***condition.release()***, consumer then fetches request from queue and starts executing and calls ***condition.notify()***.

5) Consumer always calls ***condition.release()*** after processing.

Producer View:

1) Similarly, we use ***if condition.acquire()*** to check out if we can acquire the underlying lock.

2) If the size of queue is limited, you can check out whether the queue is full.

If yes, then you call ***condition.wait()*** and ***condition.release()*** just like what consumer does.

Another choice is that simply letting the producer push requests into queue constantly.

3) Once producer can acquire the lock, it pushes request into the queue, and then calls ***condition.notify()*** and ***condition.release()*** in order.

3. Configurations

In this section, we mention several static parameters you should know.

3.1 Static Parameters List

1) Toy Warehouse in “server.py”

- *price, stock*

2) Socket in “server.py”

- *(IP address, port number)*

3) Server Console

- please type in: *size of thread pool*

4) Socket in “client.py”

- *(IP address, port number)*

5) Client Console

- please type in: *toy name you want to query number of request*

4. Reference

[1] Producer-consumer problem in Python
(https://www.agiliq.com/blog/2013/10/producer-consumer-problem-in-python)

[2] Wikipedia: Producer-consumer problem
(https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem)

[3] Python: Thread-based parallelism
(https://docs.python.org/3/library/threading.html)

[4] UMass COMPSCI 377 Course Page, Lab 2: Threads and Synchronization
(https://lass.cs.umass.edu/~shenoy/courses/fall16/labs/lab2/lab2.html)

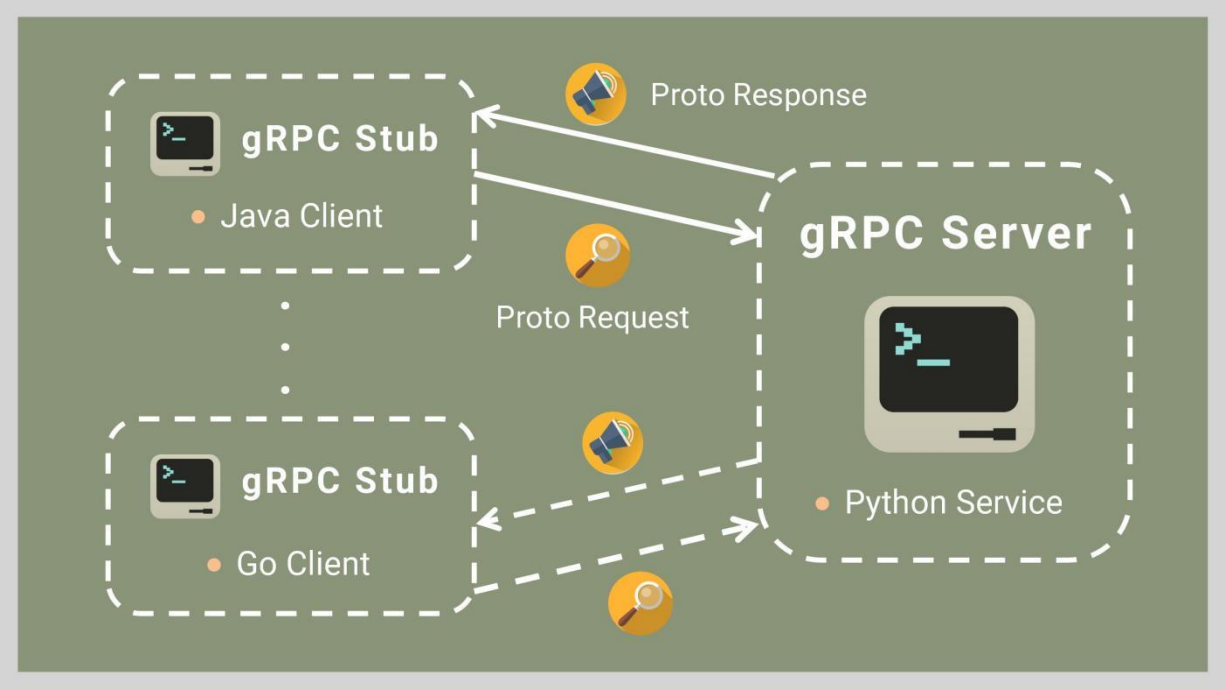
Part 2 - Implementation with gRPC and Built in Thread Pool

1. Framework - gRPC

Now let's talk about why most programmers use high-level abstractions such as gRPC and built-in thread pool mechanisms to write client-server application.

1.1 Overview

In distributed systems, the goal of a a remote procedure call (RPC) is making distributed computing look like centralized computing. gRPC is a modern high performance RPC framework that can run in any environment. As we learned in the class, service-based architecture is applied in gRPC. The main idea behind it is defining a service, specifying the methods that can be called remotely with their parameters and return types.



The gRPC methods we used to create connection are shown below:

1) Create Client

- grpc.insecure_channel(): creates an insecure channel to a server.

2) Create Server

- grpc.server(): creates a server with which RPCs can be serviced.

- grpc.start(): start your server.

- grpc.add_insecure_port(): opens an insecure port for accepting RPCs.

- grpc.wait_for_termination(): blocks current thread until the server stops.

1.2 Protocol Buffers

Protocol buffers are super important in gRPC framework. They are ideal for any situation where you need to serialize structured, record-like, typed data in a language-neutral, platform-neutral, extensible manner. They are most often used for defining communications protocols and for data storage.

Everyone should create a ".proto" file in source code at first. Here we briefly explain a few of important protocol buffer language we use.

1) Syntax Version

We go with syntax proto3.

2) Message Structure

Define a message type can help us describe a message format of request or response clearly. Here are 3 messages we defined in our ".proto" file.

- *message RequestBody*: define a format of request for both query and buy.

Here it can be a string of toy name.

- *message QueryResponse*: define a format of response for query method

Here, it can be two integer type results.

- *message BuyResponse*: define a format of response for buy method

Here, it can be an integer type result.

- *message Toy*: define basic properties for toys including name, stock and price.

3) Service Definition

There are 2 services we defined in our ".proto" file.

- *rpc Query*: Considering the message type we defined previously, Query method takes *message RequestBody, and returns message QueryResponse.*

- *rpc Buy*: Considering the message type we defined previously, Buy method takes *message RequestBody, and returns message BuyResponse.*

2. Built-in Thread Pool

2.1 ThreadPoolExecutor

In this part, we use a built-in dynamic thread pool support ThreadPoolExecutor, which is an Executor subclass that uses a pool of threads to execute calls asynchronously. Now it is very convenient for us to create a thread pool, that is simply call ThreadPoolExecutor(), and then configure the maximum number of threads in the pool.

2.2 Concurrency & Synchronization

Similarly, we also have to use appropriate synchronization methods on the product catalog in this part, since querying and buying will read from and write to the product catalog, which make it a shared data structure.

Query():

1) Try to acquire the underlying lock using ***condition.acquire().***

2) After checking out if the query item exists in the database, then call ***condition.release()***.

Buy():

1) Try to acquire the underlying lock using ***condition.acquire().***

2) Check out if the item bought by client exists in the database, and also check if the item is out of stock, then call ***condition.release()***.

Note that whatever the item bought by client looks like, we should always call ***condition.release()*** at the end, as it is locking the database.

3. Configurations

3.1 gRPC Generate Command

Install package: $ pip install grpcio_tools

Generate the Python code: after $cd to the directory where you put your files,

$ python -m grpc_tools.protoc --proto_path=. --python_out=. --grpc_python_out=. protobuf.proto

3.2 Static Parameters List

1) Toy database in “server.py”

- name, *price, stock (i.e message Toy)*

2) Connection in “server.py”

- *(IP address, port number)*

3) Server Console

- please type in: *size of thread pool*

4) Connection in “client.py”

- *(IP address, port number)*

5) Client Console

- please type in: *method you call, toy name, number of request*

4. Reference

[1] gRPC Documents: Basic Tutorial

(https://grpc.io/docs/languages/python/basics/)

[2] gRPC Python 1.44.0

(https://grpc.github.io/grpc/python/grpc.html)

[3] concurrent.futures — Launching parallel tasks

(https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor)

[4] Google Developers: Protocol Buffers

(https://developers.google.com/protocol-buffers/docs/proto#services)

[5] Python: Thread-based parallelism

(https://docs.python.org/3/library/threading.html)