Project 3 GRPC Parin Patel

Description

As per instruction provided in the README.md , We have implemented : - You are going to build a store (You can think of Amazon Store!), which receives requests from different users, querying the prices offered by the different registered vendors. - Your store will be provided with a file of `<ip address:port>` of vendor servers. On each product query, your server is supposed to request all of these vendor servers for their bid on the queried product. - Once your store has responses from all the vendors, it is supposed to collate the `(bid, vendor_id)` from the vendors and send it back to the requesting client.

How to build

You will require few requirements installed as mentioned in setup.md :

1. `cmake` - For building C/C++ applications 3.10+
2. `vcpkg` - Package Manager for C/C++ libraries
3. `protobuf` - Google Protocol Buffers
4. `gRPC` - Google's RPC framework

Note for versions

-- cmake version 3.16.3 -- grpc version --> v1.31.1 -- The C compiler identification is GNU 9.3.0 -- The CXX compiler identification is GNU 9.3.0

Important: Modify `VCPKG_HOME` variable in the `project3/CMakeLists.txt`, to where your installation of the `vcpkg` exists.

```
cd project3/
rm -rf build; mkdir build
cd build
cmake -DVCPKG_TARGET_TRIPLET=x64-linux ../
make
```

The `build/bin` folder should look like following if build succeeded

```
parin@parin-VirtualBox:[Wed Oct 21 02:59:38]:/tmp/aos3/project3/build:$ tree bin/
bin/
├── product_query_list.txt -> /tmp/aos3/project3/test/product_query_list.txt
├── run_tests
├── run_vendors
├── store
└── vendor_addresses.txt -> /tmp/aos3/project3/src/vendor_addresses.txt

0 directories, 5 files
```

How to run

From Readme.md :

- First run the command `./run_vendors ../../test/vendor_addresses.txt &` to start a process which will run multiple servers on different threads listening to (`ip_address:ports`) from the file given as command line argument.
- Start store server using :

```
    ./store ${filepath for vendor addresses} \
            ${port to listen on for clients} \
            ${maximum number of threads in threadpool}
```

optional:

- Then finally run the command `./run_tests $IP_and_port_on_which_store_is_listening $max_num_concurrent_client_requests` to start a process which will simulate real world clients sending requests at the same time.
- This process read the queries from the file `product_query_list.txt`
- It will send some queries and print back the results, which you can use to verify your whole system's flow.

Code design

When request comes :

```
run_test --> store { server --> BidClient --> Threadpool --> VendorResponse  } -->
run_vendors
```

Flow Control:

```
-> Start Async Server listening on given port
-> Create Threadpool Object with given thread count
--> Listen for requests
---> while request comes:
    ----> Once request comes , Create new object for BidClient.
    ----> Bind thread pool with BidClient this will allow bidclient to make
concerrent requests to vendors ( remember this is not async , just parellal.)
    ----> Bind Vendor Address with Bidclient.
        // Inside bidclient
        ----> if status for new client is `Create` -->  request product info from
`client` using `RequestgetProducts` , and update status toprocess
        ----> if status is process ---> Then create channels for communicting to
vendors and create future task to query vendor information/bid info (This part is
creates object for Vendor Class to be used inside threadpool) and pushes the task
into threadpool list.
            // Threadpool
```

```
            ---> Since init , while worker queue is empty , wait for upcoming
work.
            ----> if workqueue is not empty , Pop the first task from list and
excute.
                /// here We will assume Threadpool is only doing same below task

                /// Async Calls to vendors ... finally async part.
                ----> Using VendorClient object, first tasks calls all vendors
asyncly about bids and move forward.
                ----> now it starts waiting for responses from vendors using
promise. and once result comes it returns the promise back to caller , in our case
BidClient.
            ---> Now that task is done , thread pool goes back into pool after
reseting all variables , waiting for work.
        ----> Assuming promise is fullfiled , client will have bids from vendor ,
format in required format and return them in grpc call. and mark status done.
        ----> if status is done ---> clean up the client.
```

Code introduction

store.cc

```
class AsyncStore {
public:
  ~AsyncStore() {}

  /*
  Init AsyncStore class , which calls readvender_address and does soem
  validation with port.
  */
  AsyncStore(const std::string &path, int port) {}

  void run(threadpool *pthreadpool) {}

private:
  std::vector<std::string> vendor_address;
  std::string server_address;
  store::Store::AsyncService service_;
  std::unique_ptr<grpc::ServerCompletionQueue> queue;
  std::unique_ptr<grpc::Server> server_;
  threadpool *pool;

  /**
   * Read Provided Vendor Address file
   * */
  void read_vender_address(const char *const vendor_file_path) {}
  // Starts server on given port
  void start_server() {}

  void shutdown() {}
  // Handle RPC calls Asycly
```

```
  void HandleRpcs() {}
};
```

thread pool

```
class threadpool {

public:
  // validate thread pool and initailze all required private variables
  explicit threadpool(int thread_count);
  // Start the workers and enroll them in the pool.
  void init();

  // Enqueue tasks with function signature and given arguments.
  template <typename Func, typename... Args>
  auto enqueue(Func &&f, Args &&... args);

private:
  int thread_count;
  std::vector<std::thread> thread_list;

  //  List to vars for managing race condition around queue
  std::queue<std::function<void()>> queue{};
  std::mutex work_queue_lock{};
  std::condition_variable wait_for_queue{};

  // Function which pops task from queue when possible and execute function.
  [[noreturn]] void do_work();
};
```

BidClient

```
class BidClient
{

public:
    // init Bidclient and Assosiated variables and start client to procceed state
    BidClient(store::Store::AsyncService* service,
grpc_impl::ServerCompletionQueue* pQueue);
    // Entrypoint for Client
    void Procceed();
    // Bind Thread pool class object
    void Assign_ThreadPool(threadpool* pool);

    threadpool* pool;

    // Bind Vendor Address to grpc channels to be used by bidclass
    void bind_address(std::vector<std::string> vendor_address);
```

```
private:
    store::Store::AsyncService* private_service;
    grpc::ServerCompletionQueue* private_queue;
    grpc::ServerContext private_context;

    store::ProductQuery client_request;
    store::ProductReply client_response;
    grpc::ServerAsyncResponseWriter<store::ProductReply> client;

    // Let's implement a tiny state machine with the following states.
    enum QueryStatus
    {
        CREATE,
        PROCESS,
        FINISH
    };
    QueryStatus status_; // The current serving state.

    bool Create();
    //create channels for communicting to vendors and create future task to query
vendor information/bid info (This part is creates object for Vendor Class to be
used inside threadpool) and pushes the task into threadpool list.
    bool Run();
    // Update state and cleanup
    void Done();

    // metavarible for race condition
    std::mutex vendor_lock{};
    std::condition_variable wait_for_vendor_lock{};

    std::vector<std::shared_ptr<grpc::Channel>> vendor_channels;
    // function which cordinates for Async request using VendorClass
    std::vector<VendorResponse::response> list_bid(std::string basicString);
    std::vector<VendorResponse*> connect_with_vendors();
    std::vector<VendorResponse::response> AsyncRequestQuery(std::string
basicString);
};
```

VendorResposne

```
class VendorResponse
{
public:
    // Data structure to hold incoming vendor data.
    struct response
    {
        double price{};
        std::string v_id;
        std::string product;
```

```
    };

    explicit VendorResponse(std::shared_ptr<grpc::Channel> ptr)
        : vendor_stub(vendor::Vendor::NewStub(ptr)){};

// Ask vendor for bid and move forward without waiting for response
    void Async_Vendor_assemble_bid(std::string bidQuery);

//  Wait for response to come and then return request.
    response Async_vendor_response_bid();

private:
    std::unique_ptr<vendor::Vendor::Stub> vendor_stub;
    grpc::CompletionQueue grpc_client_queue;

    // struct for keeping state and data information
    struct AsyncClientCall
    {

        vendor::BidQuery query;
        // Container for the data we expect from the server.
        vendor::BidReply reply;
        // Context for the client. It could be used to convey extra information to
        // the server and/or tweak certain RPC behaviors.
        grpc::ClientContext context;
        // Storage for the status of the RPC upon completion.
        grpc::Status status;
        std::unique_ptr<grpc::ClientAsyncResponseReader<vendor::BidReply>>
response_reader;
    };
};
```

References

1. Asynchronous-API tutorial – gRPC
2. Chapter 1. Boost.Bind - 1.74.0
3. grpc/greeter_async_server.cc at v1.32.0 · grpc/grpc · GitHub
4. cplusplus.com - The C++ Resources Network
5. cppreference.com
6. multithreading - Thread Pool C++ Implementation - Code Review Stack Exchange
7. c++ - C++11 Dynamic Threadpool - Stack Overflow
8. c++ - Thread pooling in C++11 - Stack Overflow
9. c++ - Anonymous std::packaged_task - Stack Overflow
10. C++11 Multithreading – Part 10: packaged_task Example and Tutorial – thispointer.com
11. c++ - terminate called recursively - Stack Overflow