gRPC + GSoC

student proposal

Enable building of gRPC Python with Bazel

Naresh Ramesh

Github: ghostwriternr

Email: ghostwriternr@gmail.com

Table of Contents

1. About me

I am Naresh, a final year student from Indian Institute of Technology, Kharagpur pursuing a dual degree (Bachelors + Masters) in Computer Science and Engineering. I have done a couple of internships so far, the latest being a Software Engineer Intern at Intuit, India. I'm currently an organisation maintainer of MetaKGP, an Open Source community with over 400 students from my college. When not programming, I write poetry or read books.

Relevant information:

- GitHub handle: ghostwriternr
- Email address: ghostwriternr@gmail.com
- Personal site: https://ghostwriternr.me/
- Contact number: +91-7872833729
- Timezone: IST (UTC +05:30)
- Resume: PDF

2. Gaining basic knowledge

📢 **Note**: Skip this section if short on time. This section is mostly collecting my own thoughts on the background study before beginning work on the project. I've added typography cues to skim through this section easily.

2.1 gRPC and Protobuf

My first step was to go through the C++ and Python tutorials available online on gRPC's official site and made sure I was at least comfortable with using the final tools provided to the end users. I also learnt about the Protobuf specification and how (& why) it fits into gRPC.

Getting familiar with the codebase

I took a couple of days to read the code, try and understand how it was organised. I did so by trying to build the project, first the core, then gRPC Python. I **followed the chain of files being used in the build process to make sure I understand at a high level how it works**. Running tests took unbelievably long on my machine (~40 minutes), so I used the same approach to figure out **how to run the tests without having to rebuild core every time** (which took most of the time).

Issues I ran into (no longer relevant)

I briefly had some issues with running the Docker containers due to some misconfiguration on my Arch setup, but most of my troubles were because I was inside my university's network which is **restricted by a proxy (but this isn't a concern during the summer since I'll be at home)**. Specifically, when I was trying to build the Python artifacts, I noticed that `yum` simply wouldn't work on the `manylinux1` base Docker image, no matter how many ways I define the proxy configurations (Known issue with CentOS 5). So, I set up a **remote server (Ubuntu 16.04) on DigitalOcean**, where I then started doing all my work (I also learnt vim along the way, which was a lot of fun!)

2.2 Bazel

I had never used Bazel before I took up this project and started exploring it's uses. I learnt that Bazel was the open-source version of Google's highly praised internal build tool, Blaze. Picking up and using Bazel was pretty straightforward. Bazel's underlying concepts are all well explained on the official documentation.

Hermeticity

Hermeticity was a pretty important and interesting concept I came across in a couple of **talks about Bazel**, and later on **Bazel's test encyclopedia**. I've been trying to understand what changes might be needed to the current gRPC code to enforce hermeticity in the proposed build process, but that might take longer than the short time I have before proposal submission. But it will definitely be a thing I'll keep in mind if and when I get to work on the project.

Bazel + Python

When it came to the case of Python, I **talked to a few ex-interns and current employees** I know from Google and tried to understand what a **typical workflow is when working with a Python codebase**. They were very helpful in getting my head around the concept of a 'binary' (or the lack of it) in the case of Bazel + Python and how py_binary & py_test are to be used.

2.3 Cython

I had seen cython code before, but never used, built or wrote cython code myself. So to answer my own questions of what is Cython and how does it work, I started reading and found this lovely blog post by Thomas Nyberg **explaining what are cpython extension modules**. A couple of Stack Overflow answers then further helped me fully understand **how cython makes writing and maintaining this so much easier than writing native C extensions** (which can still be occasionally used for highly performance critical functions).

3. Contributions to grpc/grpc

My first contact with gRPC was on 25th February, 2018, when I first cloned the repo to see if I am able to wrap my head around the code. I've since joined in on both gRPC's community video meetings (02/03 and 15/03) to get to know the community better and also get clarifications on the questions I had.

The contributions below are as of 17th March, 2018 - 1800 IST. Sorted by latest first order.

Merged Pull Requests: 2

Create README for gRPC Python reflection package #14657

Update Dockerfiles for python artifacts to use latest git version #14588

Open Pull Requests: 2

Make logging after success in jobset more apparent #14652

Update logging in Python to use module-level logger #14557

Opened Issues: 1

Changes to build_artifact_python.sh not reflected when building artifacts #14654

4. The project

4.1 Original project idea

"Bazel is the designated replacement for our constellation of crufty build scripts, but it's still under active development itself. Up for a challenge? gRPC Python could easily be the most complex codebase to be built with Bazel."

4.2 My understanding of the problem

gRPC Python looks to be one of the most popular and complex codebases to be built with Bazel. The reasons for the complexity are manifold, but can be summarised quickly as below:

- gRPC Python **depends on a lot of native code** (written in C) to be built. The native code is a combination of gRPC core and a set of external dependencies.
- gRPC Python **uses Cython** (compiled language that generates CPython modules and makes it easy to work with them), which as of today, isn't supported by Bazel.
- Bazel in itself is **still a huge moving target** in the context of Python. It **currently only supports some basic functionality** that most certainly isn't enough for most Python projects as is, without adding custom rules.
- Making builds **reproducible on every environment** is a difficult problem and is not one that Bazel for Python has solved (yet).

4.3 Why Bazel?

Bazel is a great replacement for the following reasons (*can be* skipped if already familiar with Bazel):

- Bazel caches all the work it does so that it can **incrementally build** only changed code between builds. In most projects, `setup.py` might spend a majority of the time building native code, which can be completely avoided with Bazel.
- Similarly, Bazel is also very smart about running tests as it **re-runs only tests for which the source files have changed** since the last run (which is most often the case).
- Bazel uses an **extensible high-level build language**, Skylark (an opinionated subset of Python), to describe build properties at a semantic level. Bazel's **declarative style of writing `BUILD` files** also enforces dependencies be clearly mentioned so as to keep it reproducible.

- Bazel’s action graph cleverly maps the relationships between it’s targets for tracking changes, and also uses it to show a nice **dependency graph for the targets** to clearly visualize (and avoid) a poor dependency network.

4.4 Bazel + Python SIG

The Bazel/Python Special Interest Group (SIG) is a great resource for discussions on Bazel + Python. Doug Greiman’s public Bazel + Python design notes document, posted on the group, is easily one of the most **comprehensive descriptions of using Bazel to build complex Python codebases**, such as gRPC. He goes into great detail to explain the many nuances of ‘building’ a Python codebase with both Python and native code depending on first/third-party C extension modules. The document also does a good job at **highlighting the general lack of agreement within the Python community** when it comes to defining a build process (as opposed to the case of languages like C++ or Java where these are fairly well understood and standardised).

4.5 Current status of Bazel in gRPC

Currently, gRPC has its own set of custom set of extended rules to build its dependencies and core (written in C). The `BUILD` file is already a massive improvement over the painstakingly long 24k line `Makefile`, and seems to be **able to build core with Bazel**! The **third_party folder** also seems to have augmented all it’s required dependencies with appropriate BUILD files to build them with Bazel.

In particular, gRPC Python has **external dependencies on boringssl, zlib and c-ares**, of which zlib and c-ares are properly augmented with Bazel `BUILD` files. boringssl-with-bazel is also augmented with one, but it is not the path included in the current `setup.py` file, for understandable reasons (We aren’t building with Bazel in `setup.py`). So, if my understanding is correct, this **solves the puzzle of building all native dependencies for gRPC Python** using Bazel.

4.6 Proposal

Broadly, there are 2 parts to the definition of a 'build' in my head (and as a consequence, 2 problems to be solved here):

- Building the code on the developer's (contributing to gRPC) machine to run/test/experiment on the same machine.
- Building artifacts to be distributed for widespread use for all the possible different combinations of Python versions and system architectures.

4.6.1 Building gRPC Python on the developer's machine

Description

Although a major understatement, this is the easier (of the two) tasks at hand here while still being daunting (and madly exciting). Currently, this task is being done using Python's recommended way of writing a `setup.py` script with setuptools and installing locally with pip.

Currently supported rules in Bazel for Python

Bazel for Python currently supports 4 rules, `py_binary`, `py_library`, `py_test` and `py_runtime`. Skipping descriptions as they are clearly defined in the documentation as well.

Setting up WORKSPACE

The current `WORKSPACE` file in project repository should suffice as it can together contain the different packages (grpcio, grpcio_health_checking, grpcio_reflection, grpcio_testing) contained in `src/python` and grpcio_tools in `tools/distrib/python/grpcio_tools`.

Building/Installing and importing pip dependencies

Problem

grpcio Python has a **few Python packages as dependencies** as well. Namely, they are: coverage, cython, enum34, futures, protobuf, six, wheel. This is where things start getting extremely tricky. Currently, Bazel doesn't support building/installing these external (pip) dependencies in a (easy) way that these can then be imported as Python packages.

Solution

A sweet **pull request to Bazel's rules_python** was merged mid-September last year that supports this functionality. It exposes **two new rules**, `pip_import` and `pip_install`. `pip_import` translates the specified requirements.txt into `@my_deps//:requirements.bzl`, which itself exposes a `pip_install` method, which then creates the dependencies' repositories. During my initial tests, these rules worked just as expected for all packages depended on by gRPC Python (and a few others I tried it with). For now, this seems to be a pretty clever implementation, has been merged into rules_python for over 6 months now and should probably work well for gRPC Python's current structure.

Concerns

I'll have to further investigate this though, for several reasons.

1. A concern that arose because of one of the following statement in Doug Greiman's design document amongst one of many suggestions to build extension modules (not exactly applicable here, but the statement is still relevant):
   *"Download a pre-compiled binary wheel from PyPI, otherwise fail. Unfortunately, many packages don't have precompiled wheels, or don't have wheels for the platforms of interest. Also, **there is a set of users (including Google) that aren't willing to download and run untrusted binaries from the Internet.**"*
   If my understanding of the code in rules_python is correct, then the `pip_import` rule checks for existing wheels, but builds it itself if unavailable.
2. **Different versions of Python need a different set of (extra) dependencies**. Although this can be fixed by adding the superset to the dependencies list, as in the long run, Bazel's caching would only be

helpful and the one time additional time shouldn't be an issue. This would also be a step in the direction of making the build hermetic.
3. This rule **seems to have issues with Python3**. Personally, I haven't been able to reproduce it, but this has been reported often.

If for any reason **we choose not to go with the above approach, there are 2 ways to go from there**:

1. **Go back to rules_python and fix this at the root**. There's been extensive and interesting discussions in the bazel-sig-python group in this regard. While **the group hasn't narrowed down on a solution yet**, but a lot of options have been discussed and vetted. With an entire community watching and helping, this summer could be a good time to add this highly requested feature to Bazel and **I'd personally go out to contribute to rules_python to get this implemented** (At this point, this project deviates from gRPC. But **is it still FOR gRPC if solving gRPC's problems is one of the primary motivators?**)
2. **Write a temporary rule around the current implementation** discussed above. Of the many reasons I can think of, lack of reproducibility could be a **good enough reason to not use pip anywhere in this build process**. Because pip recursively searches for dependencies and that could change for too many reasons. Unfortunately, the hack would have to be out of the scope of the proposal, since the reasons for going this way isn't yet clear. (I'd be up for a discussion any time!)

Either way, I'm personally very excited by all the movement going on to get Python working within the Bazel community.

Building cython with Bazel

Problem

With dependencies out of the way, comes building grpcio itself. Problem? grpcio includes a lot of cython code and there is no official bazel build rules for building Cython code.

Currently, gRPC Python uses the official recommended way for building code with cython source files. It does so with a combination of distutils (or any typical `setup.py`) and `Cython.Build.cythonize`, while using an Extension instance to define the many linker options. The `grpc/_cython/_cygrpc` folder within `grpcio` contains all the definition (`.pyd`) and implementation (`.pyx`) files (which are then included in one place as `cygrpc`).

Solution

Now to do the same with Bazel, we can write our custom rules to mimic the same functionality. The rule should be able to accept all such options and invoke `cythonize`. **Such a build rule was added to the cython repo** during late 2016. The rule defines `pyx_library`, which internally extends the `py_library` rule and also invokes `cythonize` to generate the shared object libraries for the cython source files. This has been adopted by Tensorflow too since last October.

Concerns

I don't see a big list of blockers here, except **some deep dive required to get the linker to work with Bazel's chosen Python interpreter** and use the compiled extension modules.

Building grpcio

Finally, after the previous 2 BIG steps, we should be able to use a combination of the currently supported rules to build grpcio.

Reproducible nature

Reproducibility (if using `rules_python`'s rules) would require some extra work, because system-wide installed Python packages are all available to Bazel, whether or not they are exposed as a dependency to Bazel. A workaround to this is to create a virtualenv before running `bazel run`. But that doesn't bode too well with the Bazel build process.

Alternatively, these rules could be extended to have a structure similar to site-packages alongside Bazel's runfiles directory to enforce Bazel to use only the dependencies from this folder.

Hermetic nature

In order to achieve hermeticity in gRPC Python's Bazel build, we need to enforce that all targets depend only on those dependencies which are explicitly declared as a dependency in the BUILD file.

Here, since we're building our core dependencies ourselves, we can enforce hermeticity on that part. Enforcing a site-packages like structure could also help with hermeticity for the Python dependencies.

Building grpcio_health_checking and grpcio_reflection

Since these 2 are only reference packages, building them should be more or less trivial. gRPC's own blog has an excellent post on how to use rules_protobuf to build gRPC services. This method currently works pretty well for compiling python sources for protocol buffers with gRPC support.

In order to use it in runtime though (which is something we want to do), the recommended method is again to **use `pip_import` from rules_python to import grpcio**.

4.6.2 Building gRPC Python artifacts for distribution

I consider this task to be just as (or more) important compared to the previous task. But I'd like to **mark this task as out of scope of the GSoC period**. Not only because the first task might itself take a long time, but because completing the **first task is a major prerequisite** to this and any proposal I write here could need near-complete revisions based on how the first task is implemented.

5. Project timeline (tentative)

This timeline is created because the GSoC programme recommends it, but **I'll try to move as fast as I can**. If I can do better than this, I certainly will. But this timeline is to create a '**minimum viable goal**' for myself so I can judge myself as 'successful' or not at the end of the programme.

I will also be **tracking my progress every 2 days on a shared google doc** and **write a blog post every fortnight** to share things I've learnt, my progress and what I plan to do next.

Key dates (all dates in 2018)	Task
Community bonding period begins	
April 23 - May 7	- Introduce myself and discuss the proposal with gRPC and Bazel's community - Familiarise myself better with the current codebase - Go through Bazel core to fully understand how the existing Python rules work
May 7 - May 14	- Create an actionable plan for the summer - Fully document all the existing work on the project (alongside the notes from bazel-sig-python)
Community bonding period ends	
May 14 - May 28	- Experiment with the current rules in rules_python - Write a POC for the proposal discussed during the community boding period
May 28 - June 11	- Solve 'external dependencies' for Bazel + Python (in the context of gRPC or otherwise)
First evaluation	
June 15 - June 22	- Experiment with the current rules in the current repo
June 22 - July 9	- Based on the solution for external dependencies, build cython with Bazel - Solve the problem of correctly linking the build C extensions

Second evaluation	
July 13 - July 27	- Link everything together and be able to build gRPC Python on a developer's machine
July 27 - August 6	- Iron out all bugs - Document all progress in full detail
Final evaluation	

6. Why gRPC and why this project?

I have recently been **heavily exploring microservices and containers**, and had come across gRPC several times as the modern RPC framework that fits perfectly within most such architectures. I have also been **contributing to a lot of small Open Source repositories** over the past few years, but always longed to **contribute to a larger one**. This is my **first time applying to GSoC** and I nearly jumped seeing gRPC on the list, as the impulse towards the chance for contributing to gRPC was instant.

I'm a **big fan of Python** (and recently golang) and was pretty enthralled to see Python being supported so well by gRPC and also being the language of choice for the build process right now. But right from my first few days of building the project and running tests, I noticed that **improving the build process with Bazel to make it easier (and potentially speeding it up significantly) can be a super exciting contribution** that would enable both current and future contributors to build, test and work faster. Besides, I believe this would also force me to go through the codebase more thoroughly and help me **better under the code, paving for making more contributions even after the GSoC period**! Hence, this project.

7. Why me?

I started working on this project because of the above mentioned reasons, but as I kept learning about the relevant parts, I learnt so much just from reading the code and the build process. I feel this project would further help me massively in how much I can learn about Python, gRPC and Bazel. At this current stage, the ‘GSoC student’ title means very less, but **the opportunity to interact with such an experienced community and a foray into a larger sense of Open Source is one that I will utilize to the fullest**. Any positive contributions I could add to these impact projects would only make me extremely happy.

8. Notes and additional info

I was quite happy to find out that gRPC’s Nathaniel Manista was part of the bazel-sig-python group that I learnt so much from while drafting this proposal. Irrespective of whether or not I get selected, I hope to hop in on further community meetings within the SIG and absorb as much as I can and hope to use the knowledge to contribute back to the project.

9. References

- Enable building of Python gRPC library with Bazel. #8079
- Operation Purple Boa - Python + Bazel design notes
- Roadmap for Bazel + Python
- Bazel-sig-python kickstart meeting notes