N26's DNS over TLS proxy

Solution to the N26 code challenge. This Python project is a proxy that accepts TCP/UDP DNS queries and delegates them to a TLS-capable DNS server.

It's built with python3, using the new `selectors` module for I/O multiplexing. The system is configured through environment variables. If a config, for which no default value was provided, cannot be found in the environment an exception will be raised to prevent missing, required configurations from going unnoticed in production. Silent defaults are a source of bugs in production systems.

Running

There are 2 ways of running the project:

- Using docker
- As a Python script

Using docker

You can find a `docker-compose.yml` file in the root of the directory that will allow you to run the proxy with a simple command:

```
$ docker-compose up --build
```

As a Python script

Alternatively, you can run the `main.py` file in the `src` directory, providing the necessary configuration:

```
$ cd src
$ HOST=0.0.0.0 PORT=5353 DNS_HOST=8.8.8.8 DNS_PORT=853 python3 main.py
```

The software does not make use of any library outside of the standard libraries provided by the language.

Running the tests

To run the test suite you'll have to install the `nose2` module:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install --upgrade pip
$ pip install . nose2
$ python3 -m nose2
```

Manual testing

There is a DNS client included in the project that I used for manual testing during development. You can configure all the same variables as with the `dnstls` module. To see the available options, run:

```
$ python3 client.py --help
```

Keep in mind that you will need the `dnspython` module installed:

```
$ pip install dnspython
```

Concerns

Python is a great language, very expressive, that enables developers to be very productive, very quickly, as we can see in this project. With few lines of code we have a working prototype. On the downside, Python is a notable slow language that I would avoid for these kind of systems. Instead I would opt for a systems language like C or Rust, i.e. a compiled language, that makes better use of the system's resources.

My biggest concern would be the fact that far from securing the name resolution process, a custom DNS server such as this one could instead increase the attack surface. This approach does not protect the incoming, not encrypted query which could still be sniffed. And even worse, it could introduce new vulnerabilities.

That said, if we used this service internally, say inside a kubernetes cluster, in order to make sure that all outgoing DNS queries are TLS-secured, then these concerns would be mitigated.

Deployment

Deployment of this service would follow standard conventions:

- Run in multiple instances for redundancy
- Behind a load balancer
- Closely monitored
- Autoscalable

Improvements

As mentioned in the *Concerns* section, the first improvement I would introduce would be to rewrite it in a different language. The reason to use Python was to solve the given problem instead of falling victim of premature optimization.

Also:

- Write more tests
    - Due to lack of time I was not able to test as much as I'd have liked
- Support IPv6
- Allow log levels to be strings instead of ints, e.g. *debug* or *error*
- Add a healthcheck endpoint that verifies the availability of the external DNS server

- Add support for multiple DNS servers, in case one goes down
- Export metrics
    - Like the time it takes for each query
    - Or the number of queries that cannot be processed succesfuly
- Fix known issues

Known issues

The outgoing requests block the socket, i.e. while a query to the DNS server is being made, all incoming requests must wait until we get a response. While it's possible to write non-blocking code with Python, it's definitely not one of the language's strengths. Were this service ever to see production, this is a good reason, in my opinion, to migrate it to a different language.