A Fediverse/Mastodon bot for DNS queries

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

First publication of this article on 25 April 2018. Last update on of 11 January 2025

https://www.bortzmeyer.org/fediverse-bot.html

I created a bot to answer DNS queries over the fediverse (decentralized social network, best known implementation being Mastodon). What for ? Well, mostly for the fun, a bit to learn about Mastodon bots, and a bit because, in these times of censorship, filtering, lying DNS resolvers and so on, offering to the users a way to make DNS requests to the outside can be useful. This article is to document this project.

First, how to use it. Once you have a Fediverse account (for Mastodon, see), you write to @DNSresolver@mastodon.g
You just tell it the domain name you want to resolve. Here is an example, with the answer :

If you want, you can specify the DNS query type after the name (the defaut is A <https://framagit.org/bortzmeyer/mastodon-DNS-bot/issues/3>, for IPv4 adresses) :

The bot replies with the same level of confidentiality as the query. So, if you want the request to be private, use the "direct" mode. Note that the bot itself is very indiscreet : it logs everything, and I read it. So, it will be only privacy against third parties.

And, yes IDN do work. This is 2018, we now know that not everyone on Earth use the latin alphabet :

Last, but not least, when the bot sees an IP address, it automatically does a "reverse" query :

If you are a command-line fan, you can use the madonctl <https://github.com/McKael/madonctl> tool to send the query to the bot :

```
% madonctl toot "@DNSresolver@mastodon.gougere.fr framapiaf.org"
```

You can even make a shell function :

```
# Function definition
dnsfediverse() {
    madonctl toot --visibility direct "@DNSresolver@mastodon.gougere.fr $1"
}

# Function usages
% dnsfediverse www.face-cachee-internet.fr

% dnsfediverse societegenerale.com\ NS
```

There is at least a second public bot using this code, `@ResolverCN@mastodon.xyz`, which uses a chinese DNS resolver so you can see a (part of) the chinese censorship. To do DNS when normal access is blocked or otherwise unavailable, you have other solutions. You can use DNS looking glasses <https://www.bortzmeyer.org/dns-lg-usage.html>, public DNS resolver over the Web <https://dns.google.com/>, the Twitter bot @1111Resolver <https://twitter.com/1111Resolver>, email auto-responder `resolver@lookup.email`...

Now, the implementation. (You can get all the files at .) Mastodon provides a documented API. (Note that it is the client-to-server API, and it is not standard in any way, unlike the ActivityPub protocol used for the server-to-server communication. Not all fediverse programs use this API, for instance GNU Social has a different one.) You can write your own client over the raw API but it is a bit harsh, so I wanted to use an existing library. There were two techniques to write a bot that I considered, madonctl <https://github.com/McKael/madonctl> with the shell and Mastodon.py <https://github.com/halcy/Mastodon.py> with Python. I choosed the second one because a lof of nice people recommended it, and because madonctl required more text parsing, with the associated risks when you get data from unknown actors.

Mastodon.py has a very good documentation <https://mastodonpy.readthedocs.io/en/latest/>. I first create two files with the credentials to connect to the Mastodon instance of the bot. I choosed the Mastodon instance because I know it (the bot has been previously on two other instances, which have since shut down, including `botsin.space`). I created the account `DNSresolver`. Then, first file to create, `DNSresolver_clientcred.secret` is to register the application, with this Python code :

```
Mastodon.create_app(
     'DNSresolverapp',
     api_base_url = 'https://mastodon.gougere.fr/',
     to_file = 'DNSresolver_clientcred.secret'
)
```

Second file, `DNSresolver_usercred.secret`, is after you logged in :

```
mastodon = Mastodon(
    client_id = 'DNSresolver_clientcred.secret',
    api_base_url = 'https://mastodon.gougere.fr/'
)
mastodon.log_in(
    'the-email-address@the-domain',
    'the secret password',
    to_file = 'DNSresolver_usercred.secret'
)
```

Then we can connect to the instance of the bot and listen to incoming requests with the streaming API <https://mastodonpy.readthedocs.io/en/latest/#streaming> :

```
mastodon = Mastodon(
    client_id = 'DNSresolver_clientcred.secret',
    access_token = 'DNSresolver_usercred.secret',
    api_base_url = 'https://mastodon.gougere.fr')
listener = myListener()
mastodon.stream_user(listener)
```

And everything else is event-based. When an incoming request comes in, the program will "immediately" call `listener`. Use of the streaming API (instead of polling) makes the bot very responsive.

But what is `listener`? It has to be an instance of the class `StreamListener` from Mastodon.py, and to provide routines to act when a given event takes place. Here, I'm only interested in notifications (when people mention the bot in a message, a toot) :

```
class myListener(StreamListener):
    def on_notification(self, notification):
       if notification['type'] == 'mention':
            # Parse the request, find out domain name and possibly query type
            # Perform the DNS query
     # Post the result on the fediverse
```

The routine `on_notification` will receive the toot as a dictionary in the parameter `notification`. The fields of this dictionary are documented <`https://mastodonpy.readthedocs.io/en/latest/#toot-dicts`>.

For the first step, parsing the request, Mastodon unfortunately returns the content of the toot in HTML. We have to extract the text with lxml <`http://lxml.de/`> :

```
doc = html.document_fromstring(notification['status']['content'])
body = doc.text_content()
```

We can then get the parameters of the query with a regular expression (remember all the files are at).

Second thing to do, perform the actual DNS query. We use dnspython <`http://www.dnspython.org/`> which is very simple to use, sending the request to the local resolver (an Unbound) with just one function call :

```
msg = self.resolver.query(qname, qtype)
for data in msg:
    answers = answers + str(data) + "\n"
```

Finally, we send the reply through the Mastodon API :

```
id = notification['status']['id']
visibility = notification['status']['visibility']
mastodon.status_post(answers, in_reply_to_id = id,
                     visibility = visibility)
```

We retrieve the visibility (public/private/etc) from the original message, and we mention the original identifier of the toot, to let Mastodon keep both query and reply in the same thread.

That's it, you now have a Mastodon bot! Of course, the real code <https://framagit.org/bortzmeyer/mastodon-DNS-bot/blob/master/bot.py> is more complicated. You have to guard against code injection (for instance, using a call to the shell to call dig for DNS resolution would be dangerous if there were semicolons in the domain name), that's why we keep only the text from the HTML. And, of course, because the sender of the original message can be wrong (or malicious), you have to consider many possible failures and guard accordingly. The exception handlers are therefore longer than the "real" code. Remember the Internet is a jungle!

One last problem : when you open a streaming connection to Mastodon, sometimes the network is down, or the server restarted, or closed the connection violently, and you won't be notified. (A bit like a TCP connection with no traffic : you have no way of knowing if it is broken or simply idle, besides sending a message.) The streaming API solves this problem by sending "heartbeats" every fifteen seconds. You need to handle these heartbeats, and do something if they stop arriving. Here, we record the time of the last heartbeat in a file :

```
def handle_heartbeat(self):
        self.heartbeat_file = open(self.hb_filename, 'w')
        print(time.time(), file=self.heartbeat_file)
        self.heartbeat_file.close()
```

We run the Mastodon listener in a separate process, with Python's `multiprocessing` module :

```
proc = multiprocessing.Process(target=driver,
    args=(log, tmp_hb_filename[1],tmp_pid_filename[1]))
proc.start()
```

And we have a timer that checks the timestamps written in the heartbeats file, and kills the listener process if the last heartbeat is too old :

```
h = open(tmp_hb_filename[1], 'r')
last_heartbeat = float(h.read(128))
if time.time() - last_heartbeat > MAXIMUM_HEARTBEAT_DELAY:
                    log.error("No more heartbeats, kill %s" % proc.pid)
                    proc.terminate()
```

We use `multiprocessing` and not `threading` because threads in Python have some annoying limitations. For instance, there is no way to kill them (no equivalent of the `terminate()` we use. Here is the log file when running with "debug" verbosity. Note the times :

```
2018-05-02 18:01:25,745 - DEBUG - HEARTBEAT
2018-05-02 18:01:40,746 - DEBUG - HEARTBEAT
2018-05-02 18:01:55,758 - DEBUG - HEARTBEAT
2018-05-02 18:02:10,757 - DEBUG - HEARTBEAT
2018-05-02 18:02:25,770 - DEBUG - HEARTBEAT
2018-05-02 18:02:40,769 - DEBUG - HEARTBEAT
2018-05-02 18:03:38,473 - ERROR - No more heartbeats, kill 28070
2018-05-02 18:03:38,482 - DEBUG - Done, it exited with code -15
2018-05-02 18:03:38,484 - DEBUG - Creating a new process
2018-05-02 18:03:38,659 - INFO - Driver/listener starts, PID 20396
2018-05-02 18:03:53,838 - DEBUG - HEARTBEAT
2018-05-02 18:04:08,837 - DEBUG - HEARTBEAT
```