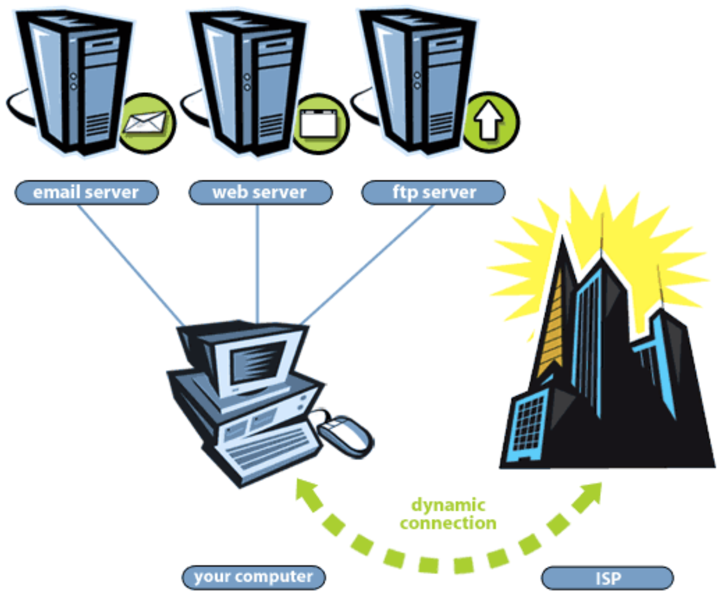
POLYTECH GRENOBLE

Open DynDNS



Lionel BOEY, Tianming GUO

RICM 4, March April, 2013

Outline

1. Context

1.1 DNS

Domain Name System (or more commonly DNS) translates Internet domain and host names to IP addresses. DNS automatically converts the names typed in the Web browser address bar to the IP addresses of Web servers hosting those sites.

DNS implements a distributed database to store this name and address information for all public hosts on the Internet. DNS assumes IP addresses do not change (are statically assigned rather than dynamically assigned).

1.2 Dynamic DNS

Dynamic DNS (DDNS) is a service that maps Internet domain names to dynamic IP addresses. DDNS serves a similar purpose to DNS: DDNS allows anyone hosting a Web or FTP server to advertise a public name to prospective users.

Unlike DNS that only works with static IP addresses, DDNS is designed to support frequenly changing IP addresses, such as those assigned by a DHCP server or ISPs. That makes DDNS a good fit for home internet subscription, which often receive dynamic public IP addresses from their Internet provider.

In general, to use DDNS, one simply signs up with a DDNS provider and installs network software on their host to monitor its IP address. For example, dyndns.com provides a free dynamic DDNS service via software that can run on Windows, Mac or Linux computers. (Discontinued on the 7th April 2014)

2. Project OpenDynDNS

2.1 Introduction

OpenDynDNS is an open source project based on the python programming language, its objective is to create a new open source DDNS solution while ensuring reliability and security in every aspect. This project was developed in Linux and can be applied for all Python supported operating systems. There are two different versions for OpenDynDNS, one is used for public networks and another is for local network.

2.2 Tools

2.2.1 Python 2.7

The Python programming language is actively used by many people, both in industry and academia for a wide variety of purposes. It powers the web apps for Instagram, Pinterest and Rdio through its associated web framework like Django, FLASK, and is used by Google, Yahoo! and NASA. Since there are still a lot of web frameworks do not support Python 3.x, Python 2.7 is this project's development environment.

2.2.2 FLASK-RESTful (Web frameworks)

Flask is a very light web framework written in python that allows provides a wide range of web server modules such as REST database, session management, Signals management, WSGI (web server gateway interface) and so on. In comparison to other more powerful web servers such as Django and Apache, Flask is actually more flexible as we choose whatever modules we need to build our own project. In this project, we only use the REST-ful modules of the webserver.

REpresentational State Transfer (REST) is a simple stateless data transfer architecture that runs on HTTP to transfer and store informations in a database. The database is built using formats such as XML or JSON(used in our project) and communications are made by HTTP requests such GET, PUT, POST and DELETE.

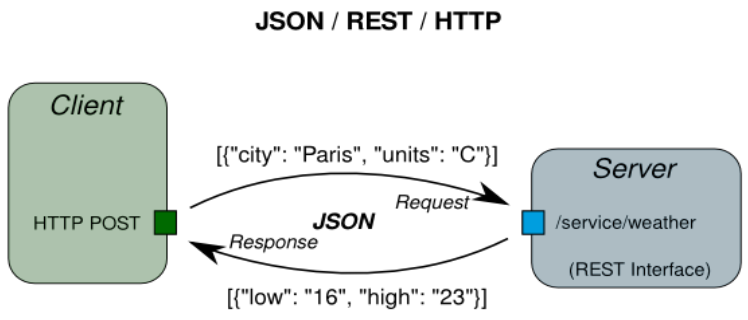


Figure 1 : An example of a REST communication

2.2.3 HTTPS and Basic Authentication

HTTPS is a basically a regular HTTP protocol implemented on top of a SSL/TLS cryptographic protocols that ensures information sent securely over the network. In our project, we enhanced the security and identity of the users with HTTPS connection and HTTP Basic Authentification, which means users will need to identify themselves with a login and a password.

2.2.4 PyZeroconf and multicast DNS

PyZeroconf is a module written in python by Paul Scott-Murphy that allows the implementation of a DNS cache and a multicast DNS (mDNS) network.

To put it simply, mDNS is a zero-configuration service that allows hosts to rapidly resolve host names of other users in a private network without a nameserver. When a needs to resolve a hostname “xyz”, it sends an IP multicast query message to the group and the target machine “xyz will send a multicast reply. All the hosts in the group will use this information to update their respective DNS cache.

2.2.5 Kivy & Python-for-Android

Kivy is an open source Python library for rapid development of mobile applications that make use of innovative user interfaces, such as multi-touch apps. A simple DDNS client was realised for Android by using kivy and Python-for-Android.

3. Public version

3.1 Concept & target audience

The public version is meant to be used in public networks (Starbucks, airport and other non-trusted network). When a service host such as a web server frequently finds itself in differents networks (hence different public IP addresses), users of this service must be able to find it by consulting the DNS server.

- Host: Check the change of its public ip address every 30 seconds, once the ip address is changed, an update will be sent to the REST server.
- REST: Receives REST updates and verify if the information is correct (id, hostname, record type etc)

  - correct updates will return a 20X status code to the host, then update REST itself and the DNS server.
  - unsuccesful updates will return a 40X status code and the update will be ignored.

Target audience for this version will include :

- Hosts who would like to provide services such as web server, mail server, file sharing services and many more but finds themselves constantly changing public IP address.
- Network administrators and developpers looking to study new technologies such as UPnP

3.2 Technologies involved

3.2.1 Security

- As previously mentioned, the public version is applied to non-trusted public network, all kinds of "normal" updates are unsafe for DNS server, so when the client wants to send an update, an authentication (username & password) and a Secure Sockets Layer(SSL) are necessary in order to successfully update the DNS information. In Flask, basic authentification is easily done by adding just a "@requires_auth" header before any class or method we would like to protect. In our project, we use the SSL feature in Flask by loading a self-generated SSL certificat (refer to Appendix). Users can subscribe for a paid certificat from a certificat authority such as godaddy.com or http://ssl.comodo.com/.

3.2.2 REST-ful server

- This server implemented by Flask serves as the gateway to the actual DNS server. Below is an small extract of the server code where HTTP requests are processed :

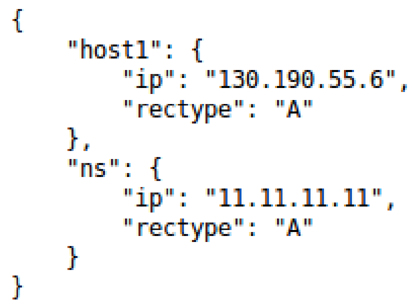
```
# A single host item
# we can GET, DELETE, or PUT a single host item
class Host(Resource):
        @requires_auth
```

```
 def get(self, host_id):
 abort_if_host_doesnt_exist(host_id)
 return HOSTS[host_id]

 @requires_auth
 def delete(self, host_id):
 abort_if_host_doesnt_exist(host_id)
 del HOSTS[host_id]
zonefilemgr.deleteHost(my_zone_file, host_id)
 //SIGHUP is sent to the bind process to reload configurations files
zonefilemgr.signalProc('named')
 return '', 204

 @requires_auth
 def put(self, host_id):
abort_if_host_doesnt_exist(host_id)
// a parser to obtain arguments from HTTP requests
 args = parser.parse_args()
host = HOSTS[host_id]
old_ip = host['ip']
new_ip = args['ip']
rectype = args['rectype']
//updates REST dictionary
modified_host = modifyDict(host_id,new_ip,rectype)
//updates bind zone files
zonefilemgr.updateHostIP(host_id, my_zone_file, new_ip, rectype)
//SIGHUP is sent to the bind process to reload configurations files
zonefilemgr.signalProc('named')
return modified_host, 201
```

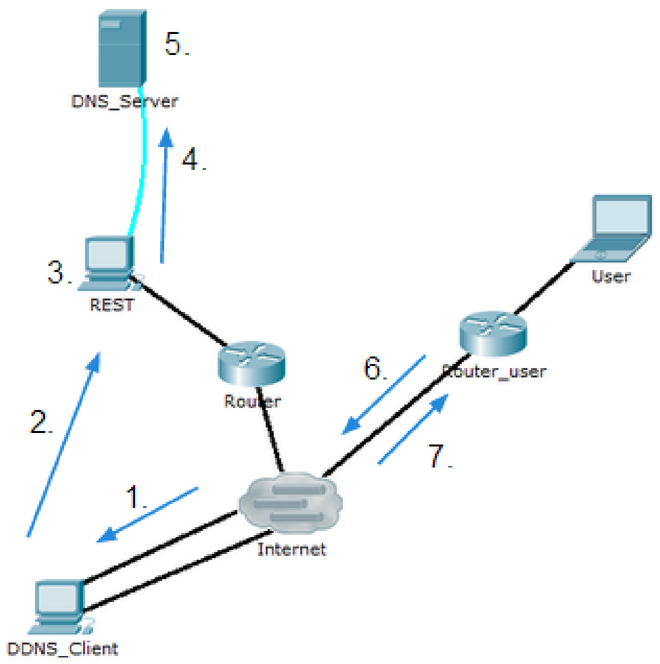
- We have created a module called "zonefilemgr.py" that provides functions to manage DNS zone files (modification, information retrieval, process signal etc).
- REST contains an identical list of DNS records stored in a JSON dictionary, the list is available after REST was launched. One or all the hosts information can be checked with a browser by https://127.0.0.1:5000/hosts/<host_id>. Once an update is completed, the new list status can be checked by refreshing the webpage. This dictionary is easily parsable by any JSON parser.



```
{
    "host1": {
        "ip": "130.190.55.6",
        "rectype": "A"
    },
    "ns": {
        "ip": "11.11.11.11",
        "rectype": "A"
    }
}
```

Figure 2 : An example of the JSON dictionary

3.3 Topology



1. DDNS_client : get public ip address a.b.c.d

2. DDNS_client : send an update to REST

3. REST : check the information of update

4. REST à DNS_server : <ddns_client @ a.b.c.d>

5. DNS_server : update the zone file

6. User à DNS_server : where is ddns.testopendyn.com?

7. DNS_Server à User : <ddns_client @ a.b.c.d>

Figure 3 : Topology of public version

4. Local version

4.1 Introduction

When a machine arrives on the local network of a company or a university, it would want to discover all the available services it can use such as web services, file hosting services, printing services and many more. In addition to being visible and accesible on the network by name, the machine would also like to resolve other host names on the network.

- Newly-arrived hosts make use of the mDNS to annouce their presence and also discover the real DNS server on the network. Resolv.conf will be updated
- DNS server will also be updated with the presence of the new host.
- The new host are able to access all the services on the network.
- Other users will be able to acces the service announced by the new host.
- The departure of the host will eventually trigger the deletion of the DNS record, thereby realising a network with rapid convergence.

Target audience for this version will include networks with dynamic hosts and clients (different machines that come and go frequently) :

- Small to medium sized companies or research labs
- Schools and universities

4.2 Technologies involved

4.2.1 Flask, HTTPS, and basic authentification

- These 3 components are reused in the network (refer to explication in the public version)
- The only difference is that the DNS server now accepts updates on LOCAL IP addresses.

4.2.2 Pyzeroconf and mDNS

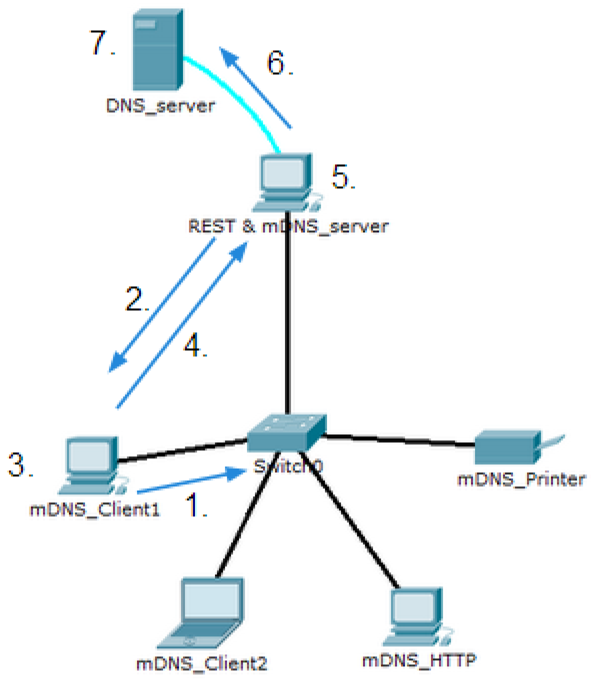
- We have studied and made use of the modules provided by the Pyzeroconf project, which can be downloaded and distributed freely on https://github.com/paulsm/pyzeroconf
- To listen to services on an mDNS network, we use a ServiceBrowser coupled with a Zeroconf object. Here is a small extract of the code used :

```
#search for real DNS server by mDNS
    try:
            find_type = "_dns._udp.local."
            // creates a Zeroconf object on all the address on this machine
            zc = Zeroconf('0.0.0.0')
            //initialise a service listener
            listener = MyListener(zc)
            //initialise a DNS service browser
            browser = ServiceBrowser(zc , find_type, listener)
            serv_ip = listener.getServAddress()
            test = 0
            //loop while browsing for DNS services
            while(not serv_ip and test<5):
                    time.sleep(1)
                    serv_ip = listener.getServAddress()
                    test += test
            print('Local IP of DNS server  : ' + serv_ip)
            #update nameserver
            addNameserver('/run/resolvconf/resolv.conf',serv_ip,'testopendyn.com')
    except Exception as err:
            print('Error in search for DNS server on the network')
            sys.exit(0)
```

- When registering a client for the first time on the mDNS network, it must first probe for a usable name and annouce a ServiceInfo which contains information about the client such as client type, full FQDN name, ip address, socket and properties. Here is an extract of the code registering a client :

```
#register client service and nameprobe on mDNS
    try:
            zc = Zeroconf('0.0.0.0')
            full_name = '%s.%s'%( base_name.split('.')[0], client_type )
            print(full_name)
            s = ServiceInfo(
                    client_type,
                        full_name, # FQDN mDNS name of server
                        server = base_name,
                        address = socket.inet_aton(client_ip),
                        port = client_port,
                properties = {"hello":"world", "dept":"ricm"}, # Setting DNS TXT records...
            )
# we register/unregister twice to prevent name conflicts by numbering client_name
            zc.registerService( s )
            client_name = zc.probeName( base_name )
            print(client_name)
            zc.unregisterService( s )
            print 'Negotiated name:', client_name
            s.server = client_name
            zc.checkService( s )
            zc.registerService( s )
    except:
            print("Error in name registration")
            zc.close()
            sys.exit(0)
```

4.3 Topology



1. client1 : member of group mDNS : I am “client1”. Who is DNS server?

2. mdns_server : member of group mDNS : I am DNS server

3. client1 : update resolv.conf

4. client1 to REST : this is my local IP address a.b.c.d

5. REST : check the information

6. REST to DNS_server : <client1 @ a.b.c.d>

7. DNS_server : update the zone file

Figure 4 : Topology of local version

5. Conclusion

5.1 Advantages and disadvantages

5.1.1 Advantages:

- Multi-platform
    - Since most of our projet is developped in Python, its can be run on any machine supporting the Python language
- Easy to use
    - After initial configurations, clients have nothing to do for the effects of DDNS to take place. When we have a confirmed use case and a stable mdns_client daemon, we could implement a more ergonomic mobile interface with Kivy.
- High flexibility and scalability
    - Our solutions provide the initial backbone for a DDNS network. Developpers can use them to implement different use cases according to their needs. For example the REST server can be transformed into a fully fledged web server such as remote domain and session management as it has the same database as the DNS server.
- If UPnP libraries and modules are finally developped and usable, hosts in the public version will be fully functional and accesible and hosts in the local version would be visible just by name from the exterior.

5.1.2 Disadvantages:

- UPnP libraries are very complicated and not yet extensively developped.
    - UPnP is a very complex protocol which involves many unstable and unsecure technologies (not just mDNS). We have tried searching for references on the internet but we only found a few half-finished projects.
- Python 3.x are not yet fully backward compatible with Python 2.x
    - Some basic functions and formats have changed in the newer version and we would have to do some translation work for our project to be used on Python 3.0

5.2 Conclusion

We can see that our preliminary version of Open DynDNS is more or less functional for real world application. However, it lacks more sophisticated features that many paid DDNS service provider offers to their clients. For future development, we could start looking into possible solutions or alternatives to UPnP which would remove the boundries between networks for all our trusted components. Also, the transition from IPv4 to IPv6 can be done and it should not be complicated.

We had faced with some difficulties during the project, such as weird bugs and we had to study very long and complicated guides on the technologies used. However, we really enjoyed this project as it has given us the opportunity to discover many useful tools and technologies. It also helps us familiarise with one of the most popular programming language in the world which is Python.

6. Bibliography and appendix

6.1 Bibliography

Flask

http://flask-restful.readthedocs.org/en/latest/

Pyzeroconf

https://github.com/paulsm/pyzeroconf

SSL certificat generation

www.akadia.com/services/ssh_test_certificate.html

Network diagnostic tools

http://www.whatsmyip.org/

Debugging and ideas

http://stackoverflow.com/

Wikipedia (concepts and definitions)

https://fr.wikipedia.org/

6.2 Appendix

- **Full project is available on GitHub**
  - **https://github.com/umpri5450/Open_DynDNS**
  - **tutorial on how to start using**
  - **our project is entirely open-source and free to distribute**

```
***Self-signed SSL certificat and key

Step 1: Generate a Private Key
   $ openssl genrsa -des3 -out server.key 1024

Step 2: Generate a CSR (Certificate Signing Request)
   $ openssl req -new -key server.key -out server.csr
   - fill in fields as you wish

Step 3: Remove Passphrase from Key
   $ cp server.key server.key.org
   $ openssl rsa -in server.key.org -out server.key

Step 4: Generating a Self-Signed Certificate
   $ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

Step 5: Usage
   - you now have a "server.key" and a "server.crt" that can be used in you SSL applications


NOTES :
To test a webserver SSL(certificates):
   openssl s_client -connect 127.0.0.1:5000
   openssl s_client -port 5000 -CApath /usr/share/ssl/certs/ -host 127.0.0.1 -prexit
```

```
$ sudo apt-get install bind9
***Bind9 zonefile configurations***
Install and set-up bind9 according to norme (guides and tutorials available online!)

Located in /etc/bind

Declare zone and reverse zone in "named.conf.local"
   zone "testopendyn.com"{
         type master;
```

```
        file "/etc/bind/db.testopendyn.com";
  };

  zone "33.33.33.in-addr.arpa" {
        type master;
        notify no;
        file "/etc/bind/db.33";
  };

Create zone file "db.<domain_name>"
  - WARNING!! : wrap A record types with marker ";hostlist" and ";endhostlist"
  - give DNS server an address

Create reverse zone file "db.33"

Check configurations files
  $ named-checkconf -z

Restart bind server
  $ sudo /etc/init.d/bind9 restart
```