Part 13
-
mDNS

Version: 2021-02-02

mDNS

If you're tired of looking up the IP addresses of devices you frequently access via remote login, SSH, and other means on your home network, you can save yourself a lot of time by assigning an easy to remember `.local` address to the device.

Why Do I Want to Do This?

Most likely your home network uses DHCP IP assignments, which means that each time a device leaves the network and returns a new IP address is assigned to it. Even if you set a static IP for a frequently used device (e.g. you set your ESP32 to always be assigned to number `192.168.1.99`), you still have to commit that entirely unintuitive number to memory. Further, if you ever need to change the number for any reason you would have to remember a brand new one in its place.

Doing so isn't the end of the world, but it is inconvenient. Why bother with memorizing IP strings when you can give you local devices easy to remember names like esp32.local or tempsensor.local?

Now, some of you (especially those of you with a more intimate knowledge of DNS, domain naming, and other network address structures) might be wondering what the catch is. Isn't there an inherent risk or problem in just slapping a domain name onto your existing network? It's important here to make note of the *big* distinction between Fully Qualified Domain Names (FQDNs), which are officialy recognized suffixes for top-level domains (e.g. the .com) and domain names that are either not recognized by the global naming/DNS system or are outright reserved for private network usage.

For example, `.internal` is, as of this writing, not a FQDN; there are no registered domains anywhere in the world that end with `.internal` and thus if you were to configure your private network to use `.internal` for local addresses, there would be no chance of a DNS conflict. That could, however, change (though the chance is remote) in the future if `.internal` became an official FQDN and addresses ending in `.internal` were externally resolvable through public DNS servers.

Conversely, the `.local` domain, has been officially reserved as a Special-Use Domain Name (SUDN) specifically for the purpose of internal network usage. It will never be configured as a FQDN and as such your custom local names will never conflict with existing external addresses.

What Do I Need?

The secret sauce that makes the entire local DNS resolution system work is known as Multicast Domain Name Service (mDNS). Confusingly, there are actually two implementations of mDNS floating around, one by Apple and one by Microsoft. The mDNS implementation created by Apple is what undergirds their popular Bonjour local network discovery service. The implementation by Microsoft is known as Link-local Multicast Name Resolution (LLMNR). The Microsoft implementation was never widely adopted thanks to its failure to adhere to various standards and a security risk related to which domains could be captured for local use.

Apple's mDNS implementation Bonjour enjoys a much wider adoption rate, has better support, and a huge number of applications for platforms big and small.

If you have computers running Apple's OS X on your network, there's nothing you need to do beyond following along with the tutorial to set things up on the ESP32 (or other Linux device) side of things. You're set to go as your computers already support it.

If you're running a Windows machine that does not have iTunes installed (which would have installed a companion Bonjour client for mDNS resolution), you can resolve the lack of native mDNS support by downloading Apple's Bonjour Printer Service helper app here (https://support.apple.com/kb/DL999?locale=en_US). Although the download page makes it sound like it's a printer-only tool, it effectively adds mDNS/Bonjour support across the board to Windows.

Use mDNS to resolve the address
In order to reach the ESP32, we need to know its IP address and, in most examples we have covered so far, we print that address to the console and then use it.
Nonetheless, in more realistic scenarios, printing the IP address might not be an option. Thus, mDNS is a protocol that allows to make the resolution of locally defined names to IPs without the need for dedicated infra-structures, such as a DNS server.

In other words, we can use a name, instead of an IP, and mDNS will allow to resolve this name into an IP address.

It's important to take in consideration that the device that is reaching the ESP32 also needs mDNS. I've tested with a Windows 10 machine, with Apple bonjour installed, and it is able to perform the resolution of the address. On older operating systems you might need to install additional software to be able to do the resolution.

Note that we don't need to install any additional library for mDNS since it is already included with the ESP32 core. The code shown here is based on the examples from the ESP32 core.

The code
We will start the code by the library includes. We will need to include the `ESPmDNS.h` library, so we have access to the mDNS related functionalities.
Additionally, we will need the `WiFi.h`, to be able to connect the ESP32 to a WiFi network, and the `ESPAsyncWebServer.h`, so we can setup a HTTP web server to run on the ESP32.

```
#include <ESPmDNS.h>
#include <WiFi.h>
```

Then we will declare two variables to hold the WiFi network credentials: network name and password.

```
const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPass";
```

Next the name of our ESP32

```
// hostname to be shown on the network
const char* hostname = "esp32";
```

Moving on, we will start by opening a serial connection and then we will connect the ESP32 to the WiFi network, using the credentials previously defined and making sure its using the hostname we assigned it

```
Serial.begin(115200);

// needed to allow setting hostname
WiFi.config(INADDR_NONE, INADDR_NONE, INADDR_NONE, INADDR_NONE);
// set hostname
WiFi.setHostname(hostname);
// act as a WiFi station/device
WiFi.mode(WIFI_STA);

WiFi.begin(ssid, password);

while (WiFi.status() != WL_CONNECTED)
{
  delay(1000);
  Serial.println("Connecting to WiFi..");
}
```

Then we will take care of setting up the mDNS responder. To do so, we simply need to call the begin method on the MDNS extern variable. This variable is an object of class MDNSResponder that becomes available when we include the ESPmDNS.h library.
As input of the begin method, we need to pass a string with the hostname that we want to be resolved to the address of the ESP32. We will set this name to "esp32".
As output, the begin method returns a Boolean value indicating if the setup procedure was successful or not. We will use that value for error checking.

```
if(!MDNS.begin("esp32")) {
     Serial.println("Error starting mDNS");
     return;
}
```

To finalize, we print the IP address to check our testing

```
Serial.println(WiFi.localIP());
```

Since we just do nothing but waiting to get a reply to a mDNS request, it means we don't need to poll any object periodically in the ESP32 main loop. Thus, for our example, it can be left empty.

```
void loop(){}
```

The final code can be seen below.

```
#include <ESPmDNS.h>
#include <WiFi.h>

const char* ssid     = "Engrie";
const char* password =  "1357924680";

// hostname to be shown on the network like in DHCP, DNS, ...
const char* hostname = "esp32";

void setup()
{
  Serial.begin(115200);

  // needed to allow setting hostname
  WiFi.config(INADDR_NONE, INADDR_NONE, INADDR_NONE, INADDR_NONE);
  // set hostname
  WiFi.setHostname(hostname);
  // act as a WiFi station/device
  WiFi.mode(WIFI_STA);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)   {
    delay(1000);
    Serial.println("Connecting to WiFi..");
  }

  if(!MDNS.begin("esp32"))   {
     Serial.println("Error starting mDNS");
     return;
  }

  Serial.println(WiFi.localIP());
}

void loop(){}
```

Testing the code
To test the code, simply compile it and upload it to your device using the ESP32 IDE. When the procedure finishes, open the Serial monitor and wait for the ESP32 to connect ot your WiFi network and print its IP address.
Then, open a command prompt in your Windows computer and ping the ESP32

```
ping esp32.local

Pinging esp32.local [192.168.1.67] with 32 bytes of data:
Reply from 192.168.1.67: bytes=32 time=9ms TTL=255
Reply from 192.168.1.67: bytes=32 time=14ms TTL=255
Reply from 192.168.1.67: bytes=32 time=22ms TTL=255
Reply from 192.168.1.67: bytes=32 time=31ms TTL=255
```

Use mDNS to advertise services
You can advertise a network service available on the ESP32 using mDNS, and get information about that service on a Python program.
For illustration purposes, the service we will setup in the ESP32 is a simple HTTP web server, using the async HTTP web server library.
For the Python program, we are going to use the zeroconf library. It can be installed with pip by using the following command:

```
pip3 install zeroconf
```

Additionally, we will make use of Python's Requests library to be able to send a HTTP request to the server hosted by the ESP32, based on the information obtained from the service. Requests can be installed with the following pip command:

```
pip3 install requests
```

The test was done using a DOIT ESP32 DevKit V1 module.

The ESP32 code
We use the previous code but add some extra coding

```
#include <ESPAsyncWebServer.h>
```

ESPAsyncWebServer.h allows to setup a HTTP web server running on the ESP32.

We instantiate an object of class AsyncWebServer, which will allow us to setup the server on the ESP32. We will pass as input of the constructor the value 80, which corresponds to the port where the ESP32 will be listening to incoming requests.

```
AsyncWebServer server(80);
```

Moving to the ESP32 setup function. To register a service, we simply need to call the `addService` method on the `MDNS` extern variable. As first input, this method receives the service name. As suggested by IDF's documentation on the lower level mDNS libraries, you can check here a list of common service names. Since our service is a HTTP server, we will pass a string with the value "`http`".
As second input, the `addService` method receives the protocol that the service runs on. In our case, our service operates over TCP, so we must pass the value "`tcp`". Alternatively we could want to register a service running over UDP, for which we would have to pass "`udp`".
As third and final input, we need to pass the number of the network port that the service runs on. In our case, as already mentioned, we are exposing our server on port 80.
Note that both the service name and protocol should be prepended with an underscore. Nonetheless, if we don't add that character to our strings, the `addService` method will add them for us.

```
MDNS.addService("http", "tcp", 80);
```

Optionally, we can also register properties for our service in the form of key value pairs. We will add two testing properties, for illustration purposes.
This is done with a call to the `addServiceTxt` method on our MDNS extern variable. As first and second inputs of this method we pass the service name and protocol, respectively. As third input we pass a string with the key and as fourth and last input a string with the value.
We can call this method more than once if we want to add multiple properties to our service.

```
MDNS.addServiceTxt("http", "tcp", "prop1", "test1");
MDNS.addServiceTxt("http", "tcp", "prop2", "test2");
```

To finalize, we are going to setup a route on the “/hello” endpoint that will listen to HTTP GET requests and answer with a simple “Hello World” message. Then we will start the server with a call to the begin method on the server object.

```
server.on("/hello", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Hello World");
});

server.begin();
```

The ESP32 main loop can be left empty since we are working with an asynchronous HTTP web server and we don’t have any additional computation to perform there.

The final ESP32 code can be seen below.

```
#include <ESPmDNS.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>

const char* ssid     = "Engrie";
const char* password =  "1357924680";

// hostname to be shown on the network like in DHCP, DNS, ...
const char* hostname = "esp32";

AsyncWebServer server(80);

void setup()
{
  Serial.begin(115200);

   // needed to allow setting hostname
  WiFi.config(INADDR_NONE, INADDR_NONE, INADDR_NONE, INADDR_NONE);
  // set hostname
  WiFi.setHostname(hostname);
  // act as a WiFi station/device
  WiFi.mode(WIFI_STA);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.println("Connecting to WiFi..");
  }

  if(!MDNS.begin("esp32"))
  {
     Serial.println("Error starting mDNS");
     return;
  }

  MDNS.addService("http", "tcp", 80);
  MDNS.addServiceTxt("http", "tcp", "prop1", "test1");
  MDNS.addServiceTxt("http", "tcp", "prop2", "test2");

  server.on("/hello", HTTP_GET, [](AsyncWebServerRequest *request){
      request->send(200, "text/plain", "Hello World");
  });
  server.begin();

  Serial.println(WiFi.localIP());
}

void loop(){}
```

The Python code

I have tested this on Python under Windows 10.
We will start by importing the Zeroconf class from the installed module.

```
from zeroconf import Zeroconf
```

We will also import the request function from the requests module. This will allow us to perform a HTTP request to the ESP32 server, after we discover its IP address.

```
from requests import request
```

Then we will create an object of class Zeroconf. We will call the constructor without passing any parameters since all the defaults are enough for our test.

```
zconf = Zeroconf()
```

Then we will create an object of class Listener, which is a class implemented by our code that will take care of handling the detection of new services of a given type.
We will check its implementation below but, for now, we will assume that its constructor takes no arguments.

```
serviceListener = Listener()
```

Then we will call the add_service_listener method on our Zeroconf class object. As first input, this method receives a string with the service type and as second it receives the instance of our Listener class.
Note that the service type is a string composed by the service name, the protocol [2] and the "local" domain, separated by a "." character. Take also in consideration that the protocol and the service name should include the underscore characters mentioned in the ESP32 code.

```
zconf.add_service_listener("_http._tcp.local.", serviceListener)
```

From this point onward, our program should now be listening for new services of the given type. If the service was already up when we run our Python code, the Listener class handler will still run once to output its information.
So, to prevent our program from ending, we will now wait for a Enter key press in the console, by calling the input function. Note that this is a blocking function but our listener will still be running on the background.
If the user clicks the Enter key, the input function will return and we assume the program should end. Thus, we need to call the close method in our Zeroconf object, This will end all the background threads and will prevent the Zeroconf instance to do more mDNS queries [3].

```
input("Press enter to close... \n")
zconf.close()
```

To finalize, we will check the implementation of the Listener class. It should have a method called add_service, with the signature defined here.
This method receives as first input the instance of the class (self), the Zeroconf instance that registered the listener, the type of the service and the name of the service.

```
class Listener:

    def add_service(self, zeroconf, type, name):
    # class implementation
```

With this information, we can call the get_service_info method on the Zeroconf object and obtain an object of class ServiceInfo, which contains the details of the service.

```
info = zeroconf.get_service_info(type, name)
```

To get the list of IP addresses associated with the service, we can call the parsed_addresses method on the ServiceInfo object. This method takes no arguments and returns an array of strings with all the addresses. In our case, we expect a single IP address for our ESP32, so the list will only have one element.

```
print("Address: " + str(info.parsed_addresses()))
```

To get the network port, we can access the port attribute of our object.

```
print("Port: " + str(info.port))
```

To get the full name of the service, we can access the name attribute. The full name will be on the following format:

```
hostname._serviceName._protocol.local.
```

Since it is a string, we can directly print it to the console.

```
print("Service Name: " + info.name)
```

Then we will print the server attribute, which corresponds has the following format:

```
hostname.local.
```

Again, we can also print it directly to the console.

```
print("Server: " + info.server)
```

We will also print the properties associated to the service, which we defined in the ESP32 code. We can obtain them by accessing the properties attribute, which corresponds to a Python dictionary.

```
print("Properties: " + str(info.properties))
```

To finalize, we are going to use the IP address obtained from the service information to do a GET request to the server.

```
address = "http://" + info.parsed_addresses()[0]+"/hello"

print("\n\nSending request...")
response = request("GET", address)
print(response.text)
```

The complete class implementation can be seen below.

```
class Listener:

    def add_service(self, zeroconf, serviceType, name):

        info = zeroconf.get_service_info(serviceType, name)

        print("Address: " + str(info.parsed_addresses()))
        print("Port: " + str(info.port))
        print("Service Name: " + info.name)
        print("Server: " + info.server)
        print("Properties: " + str(info.properties))

        address = "http://" + info.parsed_addresses()[0]+"/hello"

        print("\n\nSending request...")
        response = request("GET", address)
        print(response.text)
```

The final Python code can be seen below.

```
from zeroconf import Zeroconf
from requests import request

class Listener:

    def add_service(self, zeroconf, serviceType, name):

        info = zeroconf.get_service_info(serviceType, name)

        print("Address: " + str(info.parsed_addresses()))
        print("Port: " + str(info.port))
        print("Service Name: " + info.name)
        print("Server: " + info.server)
        print("Properties: " + str(info.properties))

        address = "http://" + info.parsed_addresses()[0]+"/hello"

        print("\n\nSending request...")
        response = request("GET", address)
        print(response.text)

zconf = Zeroconf()

serviceListener = Listener()

zconf.add_service_listener("_http._tcp.local.", serviceListener)

input("Press enter to close... \n")
zconf.close()
```

Testing the code
To test the code, start running the ESP32 code.
Once the procedure finishes, open the IDE serial monitor. You should get an output similar to below. As can be seen, we should obtain the IP address assigned to the ESP32 on the local network.

```
18:59:00.598 -> Connecting to WiFi..
18:59:00.598 -> 192.168.1.67
```

Then, run the Python code.
You should get an output similar to this.

```
Address: ['192.168.1.67']
Port: 80
Service Name: esp32._http._tcp.local.
Server: esp32.local.
Properties: {b'prop2': b'test2', b'prop1': b'test1'}

Sending request...
Hello World
```

As can be seen, the address and the port of the service match the one from the ESP32. Also, both the service name and server contain the hostname we have defined for the ESP32: the value "esp32".
We can also see that the dictionary with the service properties match the ones we have defined on the ESP32 program.
To finalize, we can confirm that we can reach the server with the information obtained from the service, since we can successfully send perform the HTTP GET request and receive a response.

Note: if you have already mDNS services running in your network, as I have, you might see an output like this. If we follow this output, you see that there are several servers replying to the request.

```
Press enter to close...
Address: ['192.168.1.64']
Port: 80
Service Name: shellyplug-6CC741._http._tcp.local.
Server: shellyplug-6CC741.local.
Properties: {b'id': b'shellyplug-6CC741', b'fw_id':
b'20201124-092420/v1.9.0@57ac4ad8', b'arch': b'esp8266'}


Sending request...
Not Found
Address: ['192.168.1.50']
Port: 80
Service Name: Volumio-Garage._http._tcp.local.
Server: volumio-garage.local.
Properties: {}


Sending request...

 <p class="browsehappy">You are using an <strong>outdated</strong> browser.
Please <a href="http://browsehappy.com/">upgrade your browser</a> to improve
your experience.</p>
 <![endif]--><div ui-view="layout" style="height: 100%;"></div><div
id="cssExposer"></div><script
src="scripts/vendor-3dafc5cf35.js"></script><script src="scripts/app-
40ba15eb14.js"></script></body></html>
```

Address: ['192.168.1.55']
Port: 80
Service Name: EPSON WF-2660 Series._http._tcp.local.
Server: EPSONWP2660.local.
Properties: {}

Sending request...

Address: ['192.168.1.39']
Port: 80
Service Name: Volumio-Keuken._http._tcp.local.
Server: volumio-keuken.local.
Properties: {}

Sending request...

You are using an outdated browser. Please upgrade your browser to improve your experience.
<![endif]--> <script src="scripts/vendor-3dafc5cf35.js"></script><script src="scripts/app-40ba15eb14.js"></script></body></html>
Address: ['192.168.1.1', 'fe80::211:32ff:febf:d275']
Port: 8000
Service Name: router._http._tcp.local.
Server: router.local.
Properties: {b'vendor': b'Synology', b'model': b'RT2600ac', b'serial': b'19B0P3N345601', b'version_major': b'5', b'version_minor': b'2', b'version_build': b'8081', b'admin_port': b'8000', b'secure_admin_port': b'8001', b'mac_address': b'00:11:32:BF:D2:74|00:11:32:BF:D2:75|00:11:32:BF:D2:76'}

Sending request...
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title id="a">The page is not found</title>
    <style>body{display:none;}</style>
    <link rel="stylesheet" type="text/css" href="/webdefault/css/error.css">
</head>
<body>
    <h1 id="b">Sorry, the page you are looking for is not found.</h1>
    <button id="c" onclick="history.go(-1)">Back</button>
    <script src="/webdefault/js/locale.js"></script>
    <script src="/webdefault/js/error.js"></script>
</body>
</html>

Address: ['192.168.1.18']
Port: 5000
Service Name: Storage3._http._tcp.local.
Server: Storage3.local.
Properties: {b'vendor': b'Synology', b'model': b'DS1813+', b'serial': b'1340LON002107', b'version_major': b'7', b'version_minor': b'0', b'version_build': b'41222', b'admin_port': b'5000', b'secure_admin_port': b'5001', b'mac_address': b'00:11:32:20:55:af|00:11:32:20:55:b0|00:11:32:20:55:b1|00:11:32:20:55:b2'}

```
Sending request...
<!DOCTYPE html>
<html>
    <body>
        <input type="hidden" id="http" name="http" value="5000">
        <input type="hidden" id="https" name="https" value="5001">
        <input type="hidden" id="prefer_https" name="prefer_https"
value="false">
    </body>
    <script type="text/javascript">
        var protocol=location.protocol;
        var port=location.protocol === "https:" ? 5001 : 5000;
        var URL=protocol+"//"+location.hostname+":"+port+"/";
        location.replace(URL);
    </script>
</html>
Address: ['192.168.1.49']
Port: 80
Service Name: Volumio-Living._http._tcp.local.
Server: volumio-living.local.
Properties: {}


Sending request...

 <p class="browsehappy">You are using an <strong>outdated</strong> browser.
Please <a href="http://browsehappy.com/">upgrade your browser</a> to improve
your experience.</p>
 <![endif]--><div ui-view="layout" style="height: 100%;"></div><div
id="cssExposer"></div><script
src="scripts/vendor-3dafc5cf35.js"></script><script src="scripts/app-
40ba15eb14.js"></script></body></html>
Address: ['192.168.1.124']
Port: 80
Service Name: shellyswitch25-68C63AFA3734._http._tcp.local.
Server: shellyswitch25-68C63AFA3734.local.
Properties: {b'id': b'shellyswitch25-68C63AFA3734', b'fw_id': b'20201124-
091406/v1.9.0@57ac4ad8', b'arch': b'esp8266'}


Sending request...
Not Found
Address: ['192.168.1.67']
Port: 80
Service Name: esp32._http._tcp.local.
Server: esp32.local.
Properties: {b'prop2': b'test2', b'prop1': b'test1'}


Sending request...
Hello World
```