SIC Lab
Programmatic Bluetooth

version 1.0

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version

1 Introduction

Bluetooth is a wireless technology originally designed and first introduced in the late 90s that is meant to replace cables. The very first version was created to replace RS-232 serial cables, and that legacy still lives within Bluetooth. After many iterations and evolutions, Bluetooth is used today to connect a wide range of personal devices (e.g., earbuds, smartbands/smartwatches, home sensors, smartplugs, smartlamps, etc.). In the previous practical lecture we have introduced and explored BlueZ, the default Bluetooth stack built into the Linux Kernel. Linux applications and services can interact with Bluetooth devices through the BlueZ stack, using the provided management applications (such as bluetoothctl) or programmatic interfaces (D-Bus or libraries). In this guide we will explore how to perform the same actions of the previous class using our own Python code using the PyBluez module and D-Bus integration with pydbus. Then, we will create a simple chat application that uses Bluetooth to send messages across devices.

2 Setup

We will start by installing the `python3-bluez` and `python3-pydbus` package in Kali. You can do it using the APT package manager:

```
$ sudo apt install python3-bluez python3-pydbus
```

– **OR** –

if you are using a different Linux distribution or want to use Python virtual environments you may install the module via PIP:

```
$ pip install pybluez
$ pip install pydbus
```

3 Doing bluetoothctl operations in Python code

We will start by coding in Python methods that perform the same operations available via the `bluetoothctl` utility. We will start by scanning for devices, listing the profiles available in each device, performing pairing/bonding to a device, and then connecting to a device (using a profile).

3.1 Scanning for devices

In order to use a Bluetooth device we must first be able to find it with our adapter to make sure we have wireless signal, and then retrieve the identifiers (e.g., MAC address) and available profiles to connect to.

We will start with a basic code for device scanning that should be able to find nearby Bluetooth devices:

```
import bluetooth

def scan_bluetooth_devices():
    try:
        print("Scanning for Bluetooth devices...")
        nearby_devices = bluetooth.discover_devices(lookup_names=True)
        print(f"Found {len(nearby_devices)} devices.")
        for addr, name in nearby_devices:
            print(f"Device: {name}, Address: {addr}")
    except bluetooth.BluetoothError as e:
        print(f"Bluetooth error: {e}")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    scan_bluetooth_devices()
```

Test the code and check the results:

1. How many Bluetooth devices did you find?
2. Can you identify *your* Bluetooth devices in that list?
3. Does a rescan yield different results?

Exercise: Please enhance the previous code by setting a reasonable timeout (in seconds) for the scan. Please check the PyBluez documentation on how to do this timeout.

1. Did the number of found devices change?
2. Play with the timeout value to fine tune the scan.

Are there other interesting parameters that we can play with to fine tune the scan?

3.2 Getting the device name

Let us assume that you already know the MAC address of the device you want to connect, but want to learn the current device name. You can do so by querying the device using its MAC address.

Consider the following code snippet:

```
def get_device_name(mac_address, timeout=10):
    print(f"Looking up name for MAC address: {mac_address}")
    name = bluetooth.lookup_name(mac_address, timeout=timeout)
    if name:
        print(f"Device name: {name}")
    else:
        print("Could not find device name (timeout or not in range).")
    return name
```

Exercise: Take the MAC address of one of the nearby Bluetooth devices (e.g., look at the scan in the previous exercise) and retrieve its currently configured friendly name.

3.3 Listing the available services

Now that we can find nearby Bluetooth devices with that simple code, we can enhance it to also print the available services.

```
def discover_services(device_addr):
    services = bluetooth.find_service(address=device_addr)
    if services:
        print(f"Services and UUIDs for device {device_addr}:")
        for service in services:
```

```
            print(f"|--| Service Classes: {service['service-classes']}")
            print(f"|  | Service UUID: {service['service-id']}")
            print(f"|")
    else:
        print(f"No services found for {device_addr}.")
```

Add the ability to list the services to your scan code and check the results.

1. Do all devices have at least one profile that can be used?
2. When available, look at the UUIDs closely and check if you can find any pattern. Then,

Exercise: Print the friendly names for the known services (check PyBluez documentation).

3.4 Pairing with a device

The pairing process is done at host-level, and you cannot select individual applications that are paired with the device, while others would not be paired. Once a device is paired with the host, the pairing process is already done for all applications, and additional access control can only be implemented with other Operating System (OS) level policies.

Therefore, we are going to use D-Bus to perform this action, because `PyBluez` does not implement the pairing API.

Note #1: You can unpair a device in your Linux desktop and then repair it using your own code. The bonded device will keep working on your Linux desktop.

Note #2: By mindful you should first scan for available Bluetooth devices before you can pair or connect.

```
import sys
import time
from pydbus import SystemBus

bus = SystemBus()
bluez = bus.get("org.bluez", "/org/bluez/hci0")

def pair_device(mac_address):
    # Format the DBus path for the device
    device_path = "/org/bluez/hci0/dev_" + mac_address.replace(":", "_")
    print(f"Looking for device at {device_path}")

    try:
        device = bus.get("org.bluez", device_path)

        print("Pairing...")
        device.Pair()  # This will initiate pairing

        # Wait a moment to let pairing complete
        for _ in range(10):
            props = device.GetAll("org.bluez.Device1")
            if props.get("Paired", False):
                print("Successfully paired!")
                return True
            time.sleep(1)

        print("Pairing timed out or failed.")
        return False

    except Exception as e:
        print(f"Error during pairing: {e}")
        return False

if __name__ == "__main__":
    success = pair_device(sys.argv[1])
    if success:
        print("Device paired.")
    else:
        print("Failed to pair device.")
```

3.5 Connecting to a device

Once pairing is completed and the device is bonded, you can (re)connect to a device at any time and start using its servies. Consider the following code snipped that connects to all services in a device:

```
import sys
from pydbus import SystemBus

bus = SystemBus()

def connect_device(mac_address):
    device_path = "/org/bluez/hci0/dev_" + mac_address.replace(":", "_")
    device = bus.get("org.bluez", device_path)

    try:
        print("Connecting to device...")
        device.Connect()  # This connects all available services
        print("Connected!")
        return True
    except Exception as e:
        print(f"Failed to connect: {e}")
        return False

if __name__ == "__main__":
    connect_device( sys.argv[1] )
```

Check in the Bluetooth tray that you are now connect to that device, and all services are available to your OS.

4 Developing Chat over Bluetooth

Now that we have exercised the programmatic interfaces for basic Bluetooth management (e.g., tasks done via `bluetoothctl`), we can now devise our own Bluetooth application. In this case, we will create a simple Chat application that allows clients to connect to a server using the Bluetooth radio.

Consider the following **SERVER** example:

(**Note:** make sure your adapter is set to be discoverable. You can do it with `bluetoothctl discoverable on`)

```
import bluetooth

server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
server_sock.bind(("", bluetooth.PORT_ANY))
server_sock.listen(1)

port = server_sock.getsockname()[1]
bluetooth.advertise_service(server_sock, "BTChatServer",
                            service_classes=[bluetooth.SERIAL_PORT_CLASS],
                            profiles=[bluetooth.SERIAL_PORT_PROFILE])

print(f"[*] Waiting for connection on RFCOMM channel {port}")
client_sock, client_info = server_sock.accept()
print(f"[+] Accepted connection from {client_info}")

try:
    while True:
        data = client_sock.recv(1024)
        if not data:
            break
        print(f"[Client]: {data.decode('utf-8')}")
        msg = input("[You]: ")
        client_sock.send(msg.encode('utf-8'))
except OSError:
    pass

print("[*] Disconnected.")
client_sock.close()
server_sock.close()
```

And the following **CLIENT** example:

```
import bluetooth

def run_client():
    # Discover nearby devices
    print("[*] Scanning for devices...")
    nearby_devices = bluetooth.discover_devices(duration=10, lookup_names=True)

    if not nearby_devices:
        print("[!] No devices found.")
        return

    for i, (addr, name) in enumerate(nearby_devices):
        print(f"{i}: {name} [{addr}]")

    choice = int(input("Select device to connect to: "))
    addr = nearby_devices[choice][0]

    print(f"[*] Connecting to {addr}...")

    # Look for the BluetoothChat service on the device
    service_matches = bluetooth.find_service(address=addr)

    port = None
    for match in service_matches:
        if match["name"] == "BluetoothChat":
            port = match["port"]
            break

    if port is None:
        port = 1  # fallback to port 1 (SPP)

    sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
    sock.connect((addr, port))
    print(f"[+] Connected to {addr} on port {port}")

    try:
        while True:
            msg = input("[You] ")
            sock.send(msg)
            data = sock.recv(1024)
            print(f"[Server] {data.decode().strip()}")
    except OSError:
        print("[!] Connection lost.")
    finally:
        sock.close()
        print("[*] Disconnected.")

if __name__ == "__main__":
    run_client()
```

Exercises:

1. In groups of two, please check correct operation of the chat application.
2. Improve the basic application to better identify the client that is sending a message.
3. Can you advertise a service? (hint: use `sdptool` or a D-Bus binding as the `PyBluez` depends on SDP Daemon that is no longer available in the modern `BlueZ` stack)
4. Could you create a file transfer service? (hint: see OBEX File Transfer)

5 Additional Challenges

We have performed device scanning and listing the available profiles using the `PyBluez` module and, then, proceeded to perform pairing and connecting to a profile using `pydbus`.

Could you eliminate the need for PyBluez in simple applications that just perform scanning and connection? (hint: use D-Bus)

6 Acknowledgements

Authored by André Zúquete and Vítor Cunha

7 Bibliography

- BlueZ
- PyBluez
- pydbus