応募区分：研究型論文

センサーデバイスへの仮想 IP 割り当て実験

富田　章義（とみた　あきよし）
株式会社ネットワールド
SI 技術本部 インフラソリューション技術部
ネットワークソリューション課

■ 要約

IoT で、物とインターネットがつながるが、物に IP アドレスを割り当てるためには、IP ヘッダの処理が可能な CPU を持つ必要がある。現在の小型のセンサーなどに搭載しているデバイスの CPU やメモリでは IP 処理はオーバーヘッドが大きく直接処理を行うことが出来ない。その解決策の1つとして IoT ゲートウェイでプロトコル変換を行うことにより、クラウドサーバへの直接通信などを行っていますが、デバイスの識別にパケットの中に識別子を追加するなどの処理を行わなくてはならず、従来の低レベルソケットによる通信が出来なかった。デバイス 1 つに対して、1 つの IP を割り当てることを VM やコンテナなどの仮想技術を利用せずに実現しかつプログラムの変更を最小限にする為の Ruby のライブラリを作り、通信影響などを検証した。

EIoT Ethernet to USB Adapter と Cisco PyUSB Module などのソフトウェアと組み合わせることで、1 台のアダプタ上の複数の USB デバイスに対して個別に制御を行うことが可能になると考える。

目次

1. 試作ライブラリの概要について

デバイスと IP 通信を実現する為の IoT ゲートウェイに導入するアプリケーションから、低レベルソケットを呼び出すことにより、仮想的にデバイス毎に IP アドレスの割り当てを行う。割り当てた IP を利用することで、低レベルソケットを利用した TCP/UDP 通信を行う事で以下の挙動を実現させる。

1. 1 台 IoT ゲートウェイに接続したデバイス毎に IP アドレス(以下仮想 IP)を割り当てる。
2. IoT ゲートウェイ上で、複数のアプリケーションが同一のデバイスを利用した場合、同じ IP アドレスを利用する。
3. 異なったデバイスの場合、同じ TCP/UDP Port 番号を利用して通信を可能にする。
4. デバイスに割り当てた IP アドレスに対しても Ping に対して応答する。
5. デバイスに割り当てた IP アドレスに対して割り当てを行っていない TCP/UDP ポート番号に対してパケットが到着した場合、ICMP のエラーメッセージで応答を行う。
6. 他のデバイスでオープンした TCP/UDP ポート番号と同じ TCP/UDP ポート番号にアクセスした場合でも ICMP エラーメッセージ応答で応答する。

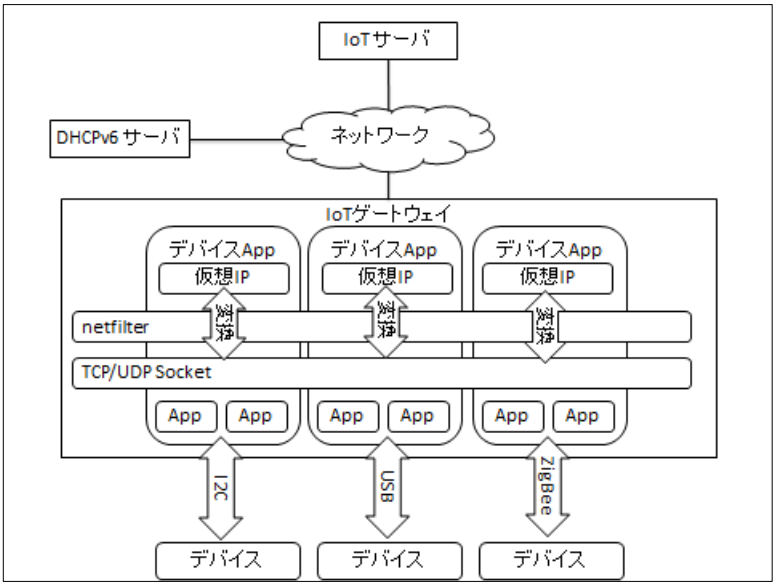

図 1　構成概要図

尚、オーバーヘッドの大きな VM やコンテナなどの仮想技術を利用すれば同様なことを実現可能だがオーバーヘッドが大きい為、今回は仮想技術を使用せずに代わりに netfilter で作成した。デバイス全てに IP を割り当てる為、IPv4 ではアドレス数が不足するので IPv6 を利用した。

2. ライブラリの動作について

2.1. IoT ゲートウェイ起動時の動作

IoT ゲートウェイが起動したときに、仮想 IP のプレフィックスには DHCPv6-PD を利用して設定を行う。この動作により、IoT ゲートウェイに対してプレフィックス値を個別に割り当てる必要やネットワーク側の機器に対してルーティングテーブルの追加を自動的に行う。
取得したプレフィックスは、ループバックインターフェースに割り当てを行うことで、DHCPv6 を利用せずにスタティックで指定したときの動作も可能にするとともに、ライブラリからプレフィックの値の参照を簡単にした。

2.2. デバイスへの仮想 IP アサイン

デバイスと仮想 IP の関連付けを行う必要があるため、以下のように socket を呼び出し時に、48 ビットのデバイス識別子を追加することで、デバイスを認識するようにした。

```
1  iotsock = IotSocket.new("001122334455", Socket::AF_INET6, Socket::SOCK_DGRAM, 0)
```

ライブラリの中では、既に同一識別子で仮想 IP を割り当てているかを判断する為に Socket の呼び出した時に内部データベースを参照することで、使用状況を確認するようにした。既に仮想 IP を割り当てているのであれば、Socket の作成処理は終了する。割り当てが無い場合は、仮想 IP アドレスの作成、netfilter へのエントリ追加、データベースへのエントリ追加をする。

仮想 IP アドレスの作成は、48 ビットのデバイス識別子を使い EUI-48to64 アルゴリズムを利用しホストアドレスを割り当てる。その後、ループバックのアドレスからプレフィックスを呼び出し一意の IPv6 アドレスを作成する。
netfilter へのエントリ追加は以下のコードで行った。主に 3 つの動作を含めており、1 つ目は、仮想 IP 宛の TCP パケットに対して、ICMP port unreachable 応答を返すエントリ追加、2 つ目は仮想 IP 宛の UDP パケットに対して ICMP port unreachable 応答を返すエントリ追加、3 つ目は TCP/UDP 以外のパケットを自装置宛に変換することで Ping 応答をするエントリ追加である。これらのエントリの追加を行うと仮想 IP アドレスが動作し外部からの通信に応答するようになる。

```
   # 新規仮想デバイスの追加
1  system("#{IP6TABLES} -t nat -N #{@conf['MAC']}_IN")
2  system("#{IP6TABLES} -t nat -A PREROUTING -d #{@conf['IPv6ADDR']} -j
   #{@conf['MAC']}_IN")
3  system("#{IP6TABLES} -t nat -A #{@conf['MAC']}_IN -p tcp
   -j DNAT --to-destination [#{LOIP6ADDR}]:4")
4  system("#{IP6TABLES} -t nat -A #{@conf['MAC']}_IN -p udp -j DNAT --to-destination
   [#{LOIP6ADDR}]:4")
```

```
5  system("#{IP6TABLES} -t nat -A #{@conf['MAC']}_IN -j DNAT --to-destination #{LOIP6ADDR}")
   system("#{IP6TABLES} -t nat -N #{@conf['MAC']}_OUT")
6  system("#{IP6TABLES} -t nat -A POSTROUTING -j #{@conf['MAC']}_OUT")
7
```

TCP/UDP の ICMP port unreachable 応答は、well-known port のうち IANA で未定義の最若番のポート(4)に NAT として処理することで実現した。

2.3. Socket のオープン時

Ruby の socket ライブラリと同じ指定を行う。ただし、割り当てる IP アドレスは IoT ゲートウェイ上にはない為、必ず無指定の[::]を利用することとした。

```
1  sockaddr = Socket.sockaddr_in(0, "::")
2  iotsock.bind(sockaddr)
```

ライブラリの中では、指定された TCP/UDP ポート番号に関わらず、ephemeral port(ポート番号 0 指定)を使いループバックインターフェースのアドレスを利用してソケットライブラリの bind を実行する。この動作により、IoT ゲートウェイで使用する TCP/UDP ポート番号と同じ番号のソケットを使用することを回避した。

自動的に割り当てられた IoT ゲートウェイ上の TCP/UDP ポート番号を調べ IoT ゲートウェイ上で通信する実ポート番号を決定する。アプリケーションから TCP/UDP ポート番号に ephemeral port が指定をした場合は、仮想 IP のポート番号を実ポート番号と同じ TCP/UDP ポート番号を利用するようにした。

仮想 IP のポート番号を使い 2 つの netfilter エントリを追加する。1 つ目は、パケットの受信時に NAT を行い IoT ゲートウェイのループバック IP アドレスとポート番号へ NAT するエントリで受信したパケットに対して処理を行う PREROUTING テーブルに追加、2 つ目はパケットの送信時に NAT を行い送信元を仮想 IP とポート番号に変換するエントリで、送信パケットに対して処理を行う POSTROUGING テーブルに追加する。

```
1  system( "#{IP6TABLES} -t nat -I #{@conf['MAC']}_IN -p #{proto} --dport #{@vport} -j DNAT
   --to-destination [#{LOIP6ADDR}]:#{@rport}")
2  system( "#{IP6TABLES} -t nat -I #{@conf['MAC']}_OUT -p #{proto} --sport #{@rport} -j SNAT
   --to-source [#{@conf['IPv6ADDR']}]:#{@vport}
```

最後にデータベースにポートエントリ情報を追記する。

2.4. TCP/UDP パケット通信時

Ruby のソケットライブラリを同じ指定を行い、特にライブラリ固有の動作を行わない。

例として、UDP パケットの送信時を掲載する。

```
1    iotsock.send(hdc_data, 0, Socket.sockaddr_in(server_port, server_addr))
```

カーネルの netfilter が動的にパケットの IP アドレスとポート番号を変更して通信する為、ライブラリ上での追加動作は行わない。

2.5. Scoket のクローズ時

Ruby のソケットライブラリと同じ指定を行う。正常処理後に仮想ポート番号を閉じ、必要に応じて仮想 IP の開放も行う。

```
1    iotsock.close()
```

ライブラリの中では、データベースを参照し、netfilter エントリの削除、ソケットライブラリのクローズ、データベースからポートエントリの削除を行う。この後、該当 TCP/UDP ポートに対してアクセスを停止する。
その後全てのポートエントリが削除されている場合、仮想 IP の netfileter エントリの削除、データベースからデバイスエントリの削除を行う。

3. 試験構成

図 1 の構成を作成し試験を実施した。
通信は TCP/UDP の切り替えが可能にし、両プロトコルで各デバイスに対して仮想的に IP を使えることを検証する。使用中の TCP/UDP ポート以外への通信に対して、ICMP エラー応答をすることを確認する。

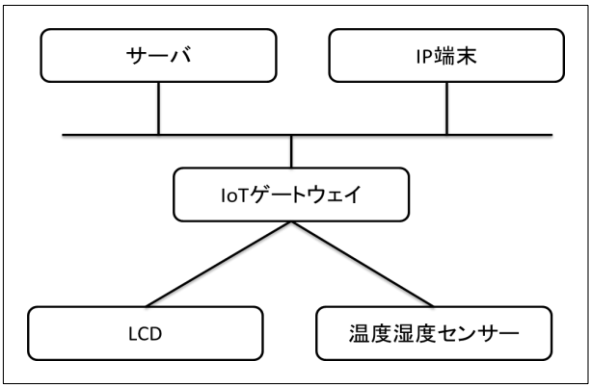

図 2　試験構成図

3.1. IoT ゲートウェイ

Linux が利用可能なカスタマイズが容易な raspberry pi を使用する。
必要となるパッケージも殆どがリポジトリからダウンロードを行ったが、iptables は NAT66 をサポー

トしていなかったので最新版に更新した。（make 処理を再度実行）

3.2. デバイス

2 つのデバイスを用意し、同時に複数の仮想 IP の割り当てを行った。
デバイス 1 台目は、TEXAS INSTRUMENTS 社 の　HDC1000 を利用する。このデバイスは、I2C 接続のセンサーになっており、温度、湿度、デバイス情報（メーカーID・製品 ID・シリアル番号）を取得可能であり、分解能・リセット・デバイスの温度を上げるテストモードの設定が出来る。
シリアル番号から仮想 IP を作成し、サーバへ 1 分間隔で温度と湿度を送信するのと同時に全ての外部装置からの分解能の変更、センサーのリセットの実行、テストモードの ON/OFF を行えるようにした。
デバイス 2 台目は、Sitronix 社の ST7032 を利用する。このデバイスは、I2C 接続の LCD になっており、文字列を表示することのみが出来る。すべての外部装置からメッセージ書き換え命令により表示するメッセージの書き換えを可能にした。

3.3. サーバ

Linux をインストールしたサーバを用意した。
IoT ゲートウェイに対して、IPv6 プレフィックスを払い出す為の DHCPv6 Server をインストールし、プレフィックスプールを作成した。また、IoT ゲートウェイの温湿センサーの仮想 IP から受信した温度・湿度データを受信してデータを蓄積するサーバ機能、蓄積されたデータから LCD の仮想 IP へメッセージを送る通知機能を作成し動作させた。

3.4. その他の IP 端末

IoT ゲートウェイの温湿度センサーの仮想 IP に対して、分解能やテストモードへ移行するための命令を送ることと LCD を上書きすることが可能なソフトウェアを作成し動作させた。

4. 試験結果

4.1. サーバとのデータ送受信

温湿度センサーと LCD の通信パケットが仮想 IP アドレスになっており、同時に複数の仮想 IP と通信が可能となった。また、サーバから各デバイスに対して Ping 応答の確認が取れ其々のデバイスが個別の IP アドレスを持っているように動作した。

4.2. ネットワーク上の端末からの送受信

その他の IP 端末から温湿度センサーの変更や LCD の書き換えが仮想 IP アドレスで行えることができる。Ping 応答や不正な TCP/UDP Port に対してパケットを送ることで ICMP Error が帰ってくることが確認できた。

5. 所感

今回実験的に作成したライブラリは従来のソケットライブラリを呼び出すときに 1 つのパラメータを追加するのみで対応を行うことが出来た。これにより、ソースコードの変更点が少なく仮想 IP を利用することが可能である。VM やコンテナを利用すれば同様に接続したデバイスに対して仮想的に IP を割り当てることが可能であり、ハードウェアと仮想化イメージとの関連付けが必要となる。また、オーバーヘッドも大きい為、ハードウェアリソースが低いゲートウェイの場合は同時に動作させるイメージ数が少なくなるが、今回のライブラリを使用すれば関連付けとオーバーヘッドの削減が可能である。ただし、既存のソースコードを流用する場合、ソケットの呼び出す部分を変更する必要がある。

デバイス IO は I2C のみで実験を行ったが、IoT ゲートウェイ上でソケットアプリケーションを使う限り特に制限は無く、Linux ベースで動作する IoT ゲートウェイであればどのような製品にも応用可能である。実験用の Ruby ライブラリ内部で iptables を実行することで netfilter の定義を変更したが、直接 netfilter のエントリを書き換えるように変更すること、ソケットオープンクローズ時の無駄なオーバーヘッドを短縮可能である。今後、この方式を拡張するのであれば、ライブラリのエンハンスが必要と考える。

以上