目次

1 はじめに

1.1 まえがき

はじめまして、エンジニアを目指している鹿さんと申します.
今回初めて本を出すことになりまして、Golang により ICMP と ARP を用いたネットワーク内のデバイス数を取得する方法を書いてみました. ICMP とは ping コマンドなどに使われるプロトコルのことで、ARP とはネットワーク上の MAC アドレスを取得するするためのプロトコルです. この本ではそれぞれ簡単な説明の後、ping や arpscan のような Linux コマンドを用いずに実装していきます.
本を執筆することも、ネットワーク系技術についても不慣れなため、不自然さや間違い等あるかもしれませんが、是非とも生暖かい目で応援よろしくお願いします.

2

ICMP・ARP の解説

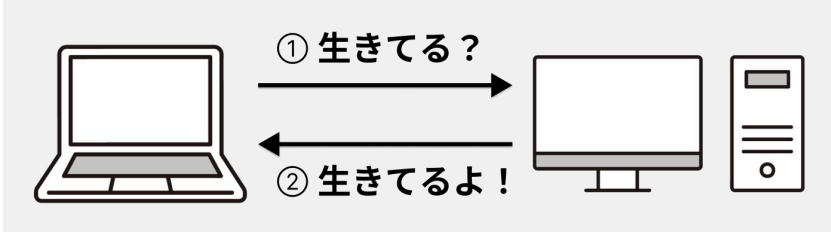
ここでは ICMP・ARP それぞれについての解説と、この 2 つのプロトコルの違いについて解説していきます。

2.1 ICMP について

ICMP の概要、パケット構成、タイプについて解説していきます

2.1.1 概要

Internet Control Message Protocol（ICMP）は、ネットワーク通信の問題を診断するためにネットワークデバイスが使用するネットワーク層のプロトコルです [1]。
ICMP には、問い合わせ用の Query とエラー発生時に発行される Error の 2 種類のメッセージが存在しており、主にデータが目的の宛先にタイムリーに到達しているかどうかを判断するために使用されています。ICMP が使われている代表的な Linux コマンドに ping があります。



▲ 図 2.1　ICMP のイメージ

2.1.2　パケット構成

パケット構成は次の図のようになっています。

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+5
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Data ...
+-+-+-+-+-
```

構成内のそれぞれの要素は次の表のようになっています。特に ping コマンドで使われ

▼ 表 2.1　ICMP パケットの要素

フィールド	ビット数	内容
Type	8bit	機能 (タイプ) の番号
Code	8bit	Type の詳細な機能コード (オプションのようなもの)
Checksum	16bit	エラーの有無を確認
Data	可変長	Type ごとに必要なデータが加わる

る Echo Type では、Date に Reply との対応付けとして使用できる識別子・番号である Identifier、Sequence Number がそれぞれ 16bit ずつ割り当てられます。

2.1.3　Type について

Type は全部で 13 個存在し、それぞれが別の機能を持っています。ping コマンドでは 0 と 8 が使われています。

▼ 表 2.2　ICMP の Type 一覧

Type 番号	機能
0	エコー応答
3	宛先不達
5	リダイレクト要求
8	エコー要求
11	時間超過
13	タイムスタンプ要求
14	タイムスタンプ応答
15	情報要求
16	情報応答
17	アドレス・マスク要求
18	アドレス・マスク応答

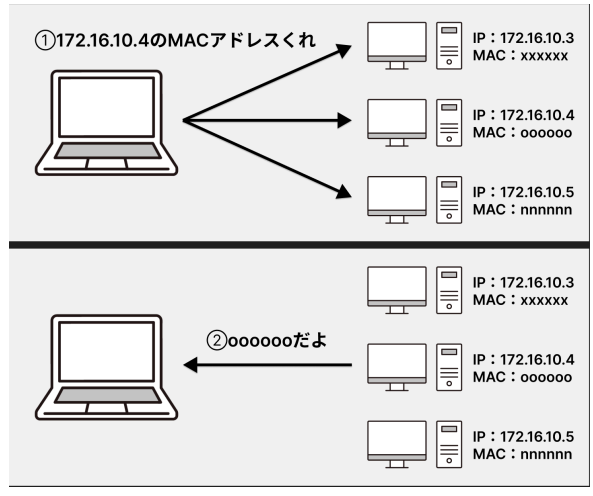
2.2 ARP について

2.2.1　概要

Address Resolution Protocol(ARP) とは、IP アドレスから MAC アドレスを求めるために利用されるデータリンク層で動作するプロトコルのことです。
要求と応答である「ARP リクエスト」と「ARP レスポンス」の 2 種類のメッセージが存在し、主にイーサネット上での物理的な通信の際に対象の MAC アドレスと IP アドレスの対応表を作成するのに使われます。なぜ IP アドレスがあるにもかかわらず MAC アドレスが必要なのかと言いうと、IP アドレスが必要となるのはネットワーク層での話であり、データリンク層では MAC アドレスを元にターゲットを特定するためです。ARP が使われている代表的な Linux コマンドとして、LAN 内の IP アドレスと MAC アドレスの一覧を表示する arpscan があります。

2.2.2　パケット構造

ARP パケットは Ethernet 形式のフレームでイーサネットヘッダによってカプセル化されています。また、イーサネットヘッダのタイプコード 0x0806 でデータ部分が ARP メッセージであることを表します [3][4]。



▲ 図 2.2　ARP のイメージ

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Hardware Type       |          Protocol Type          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Hardware Size| Protocol Size|             Opcode              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Sendr MAC address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Sendr MAC address(続き)   |       Sendr IP address          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Sender IP address(続き)   |       Target MAC address        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Target MAC address(続き)                    |
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Target IP address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

それぞれのフィールドのビット数、内容は次のようになっています。Ethernet、IPv4 の環境下では、Hardware Type、Protocol Type、Hardware Size、Protocol Size は固定値となります。

▼ 表 2.3　ARP パケットの要素 [3]

フィールド	ビット数	内容
Hardware Type	16bit	イーサネットの場合、0x0001 で固定
Protocol Type	16bit	IPv4 の場合、0x0800 で固定
Hardware Size	8bit	MAC アドレスのサイズ (Byte) MAC アドレスは 48bit=6Byte なので、0x06
Protocol Size	8bit	IP アドレスのサイズ (Byte) IP アドレス (IPv4) は 32bit = 4Byte なので、0x04
Opcode	16bit	ARP リクエスト：0x0001 ARP レスポンス：0x0002
Sender MAC address	48bit	送信元 MAC アドレス
Sender IP address	32bit	送信元 IP アドレス
Target MAC address	48bit	ターゲット MAC アドレス
Target IP address	32bit	ターゲット IP アドレス

3

ICMP、ARP を使ってネットワーク内のデバイスを取得する with Golang

3.1 ARP バージョン

まずは ARP の方から実装していこうと思います。
やることとしては、Linux コマンドである arpscan を実装する手順とほとんど同じです。Google が GitHub で公開しているサンプルプログラムを参考にしました [5]。ARP はルータを越えることができないので、同一ネットワーク内が前提です。また、今回は module の設定等については省略します。今回実装する処理の流れは以下のようになります。

1. ホストのネットワークインターフェイスを全て取得
2. 1 で取得した中から IPv4 を使用しているインターフェイスを検索
3. 2 で使用しているアドレスからそのネットワークで扱えるすべての IP を取得
4. ネットワークのすべての IP に対して ARP リクエストを送信
5. ARP レスポンスの数を計測
6. 表示

3.1.1　ホストのネットワークインターフェイスを全て取得

まずは net パッケージの Interfaces 関数によりインターフェイスのリストを取得します。ここでエラーが発生した場合は panic を発行して終了させます。

```
1 func main() {
2   // すべてのインターフェイスのリストを取得します
3 ifaces, err := net.Interfaces()
4 if err != nil {
5     panic(err)
6  }
7 }
```

3.1.2 取得したリストから IPv4 を使用しているインターフェイスを検索

IPv4 のアドレスを使用しているインターフェイスを検索し、IP アドレスとサブネットマスクを取得します。これらはインターフェイス 1 つずつ行われ、取得できなかった場合や localhost であった場合はエラーを返します。

```
1 // ARP リクエスト/リプライを使用して、
2 // 個々のインターフェイスをスキャンします
3 func scan(iface *net.Interface) (int, error) {
4   // IPv4 アドレスを取得する
5   var addr *net.IPNet
6   if addrs, err := iface.Addrs(); err != nil {
7       return 0, err
8   } else {
9       for _, a := range addrs {
10        if ipnet, ok := a.(*net.IPNet); ok {
11    if ip4 := ipnet.IP.To4(); ip4 != nil {
12                  addr = &net.IPNet{
13          IP:   ip4,
14          Mask: ipnet.Mask[len(ipnet.Mask)-4:],
15          }
16          break
17        }
18        }
19    }
20 }
```

```
21  // インターフェイスのアドレスが正しいかどうかをチェックします。
22  if addr == nil {
23   return 0, errors.New("no good IP network found")
24  } else if addr.IP[0] == 127 {
25 return 0, errors.New("skipping localhost")
26  } else if addr.Mask[0] != 0xff || addr.Mask[1] != 0xff {
27   return 0, errors.New("mask means network is too large")
28  }
29      log.Printf(
30        "Using network range %v for interface %v",
31        addr,
32        iface.Name
33      )
34  }
```

3.1.3 使用しているアドレスとマスクからそのネットワークで扱えるすべての IP を取得

IP アドレスとサブネットマスクがわかれば、接続しているネットワークで割り当て可能な IP アドレスの範囲がわかります。それらを out という net.IP 型のスライスに格納します。

```
1 // ips は、net.IPNet からすべての IPv4 アドレスを取得します
2 func ips(n *net.IPNet) (out []net.IP) {
3   num := binary.BigEndian.Uint32([]byte(n.IP))
4   mask := binary.BigEndian.Uint32([]byte(n.Mask))
5   network := num & mask
6   broadcast := network | ^mask
7   for network++; network < broadcast; network++ {
8     var buf [4]byte
9 binary.BigEndian.PutUint32(buf[:], network)
10      out = append(out, net.IP(buf[:]))
11  }
12  return
```

```
13  }
```

3.1.4　ネットワークのすべての IP に対して ARP リクエストを送信

先ほど上で ARP のパケット構造について解説した通りにデータを格納し、各 IP アドレスに対して 1 回送信します。

```
1 // writeARP は、ローカルネットワーク上の各アドレスに対する ARP リクエスト
を pcap ハンドルに書き込みます
2 func writeARP(
3   handle *pcap.Handle,
4   iface *net.Interface,
5   addr *net.IPNet
6   ) error {
7       // できる限りすべてのレイヤーのフィールドを設定します
8       eth := layers.Ethernet{
9     SrcMAC:       iface.HardwareAddr,
10         DstMAC:       net.HardwareAddr{
12             0xff, 0xff, 0xff, 0xff, 0xff, 0xff
13          },
14          EthernetType: layers.EthernetTypeARP,
15      }
16      arp := layers.ARP{
17          AddrType:          layers.LinkTypeEthernet,
18         Protocol:          layers.EthernetTypeIPv4,
19         HwAddressSize:     6,
20         ProtAddressSize:   4,
21         Operation:         layers.ARPRequest,
22         SourceHwAddress:   []byte(iface.HardwareAddr),
23         SourceProtAddress: []byte(addr.IP),
24         DstHwAddress:      []byte{0, 0, 0, 0, 0, 0},
25      }
26      // シリアライズのためのバッファとオプションを設定する
27      buf := gopacket.NewSerializeBuffer()
```

```
28      opts := gopacket.SerializeOptions{
29     FixLengths:       true,
30         ComputeChecksums: true,
31     }
32     // すべてのアドレスに対して 1 パケットを送信する
33     for _, ip := range ips(addr) {
34     arp.DstProtAddress = []byte(ip)
35     gopacket.SerializeLayers(buf, opts, &eth, &arp)
36     if err := handle.WritePacketData(buf.Bytes()); err != nil {
37     return err
38     }
39     }
40     return nil
41  }
```

3.1.5 ARP レスポンスの数を計測

readARP は ARP レスポンスを受け取り、受け取ったレスポンスの数をカウントする関数です。受け取ったパケットが空であったり、自身の送信したパケットの場合はカウントしません。この関数はチャネルによって stop を受け取るまで処理を繰り返します。

```
1 func readARP(
2   handle *pcap.Handle,
3   iface *net.Interface,
4   stop chan struct{},
5   c chan int
6 ) {
7   src := gopacket.NewPacketSource(
8       handle, layers.LayerTypeEthernet
9   )
10  in := src.Packets()
11  count := 0
12  for {
13      var packet gopacket.Packet
```

```
14      select {
15          case <-stop:
16              c <- count
17      return
18          case packet = <-in:
19      arpLayer := packet.Layer(layers.LayerTypeARP)
20      if arpLayer == nil {
21          continue
22      }
23      arp := arpLayer.(*layers.ARP)
24      if arp.Operation !=
25                  layers.ARPReply || bytes.Equal(
26                      []byte(iface.HardwareAddr),
27                      arp.SourceHwAddress
28                  ) {
29                  // 自身が送信したパケットです
30                  continue
31                  }
32              if arp.SourceProtAddress[
33                   len(arp.SourceProtAddress)-1
34              ] == 1 {
35      continue
36      }
37      count++
38              log.Printf(
39                  "IP %v is at %v",
40                  net.IP(arp.SourceProtAddress),
41                  net.HardwareAddr(arp.SourceHwAddress)
42              )
43          }
44      }
45  }
```

3.1.6　それぞれの関数を繋げて、表示

ここまでで機能的なものは完成しました。あとは一つのプログラムとして動作できるようにそれぞれの関数を繋いでいきます。main 関数、scan 関数のすでに記述した部分については「・・・」で省略します。

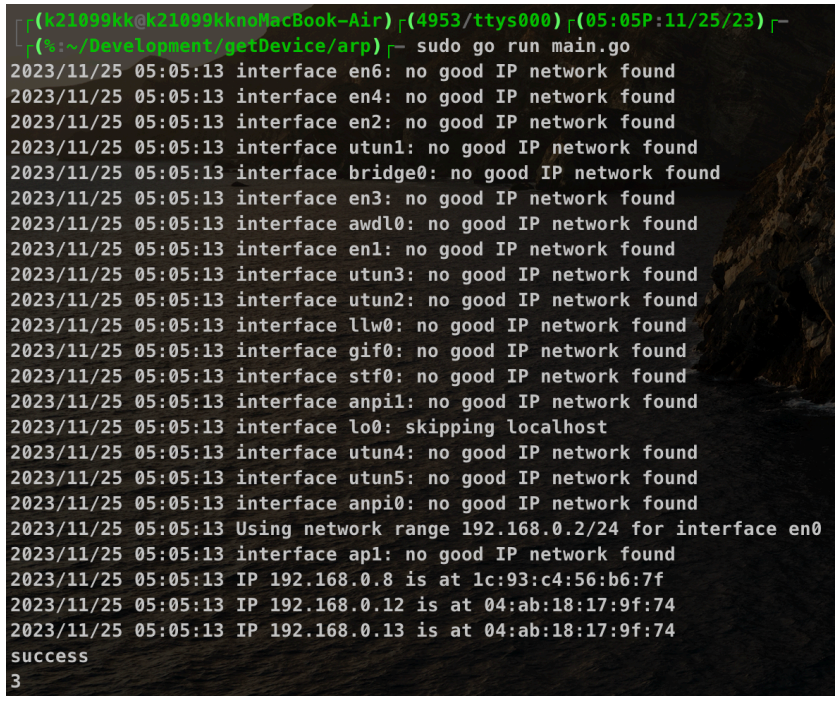
```
1 func main() {ー
2        ・・・
3   // すべてのインタフェイスをスキャンします
4   // それぞれのスキャンは、別の Goroutine で実行されます
5   // スキャンが完了すると、チャネルに結果が送信されます
6   // ふさわしくないインターフェイスの場合は、エラーが発生します
7   c := make(chan int)
8   for _, iface := range ifaces {
9       go func(iface net.Interface, c chan int) {
10          if count, err := scan(&iface); err != nil {
11              log.Printf("interface %v: %v", iface.Name, err)
12          } else {
13              fmt.Println("success")
14              c <- count
15          }
16      }(iface, c)
17  }
18  fmt.Println(<-c)
19 }
20
21 // ARP リクエスト/リプライを使用して、個々のインターフェイスをスキャンします
22 // パケットを書き込めなかった場合はエラーを返します
23 func scan(iface *net.Interface) (int, error) {
24      ・・・
25      // パケットの読み取り/書き込み用に pcap ハンドルを開きます
26      handle, err := pcap.OpenLive(
27          iface.Name, 65536, true, pcap.BlockForever
28      )
29     if err != nil {
```

```
30      return 0, err
31      }
32      defer handle.Close()
33
34      // パケットデータを読み込むために Goroutine を起動する
35      stop := make(chan struct{})
36      c := make(chan int)
37      go readARP(handle, iface, stop, c)
38      defer close(c)
39      // スキャンパケットをハンドルに書き出す
40      writeARP(handle, iface, addr)
41      time.Sleep(3 * time.Second)
42      close(stop)
43      count := <-c
44      return count, nil
45  }
```

これにて完成です！それでは実際に動かしてみましょう。

3.1.7 実行

main.go を sudo 権限で実行してください。おそらく 3.1 のような結果になったかと思います。出力の最後の方がこのプログラムで検知できたデバイス、一番最後の数字の出力が検知できたデバイス数となります。

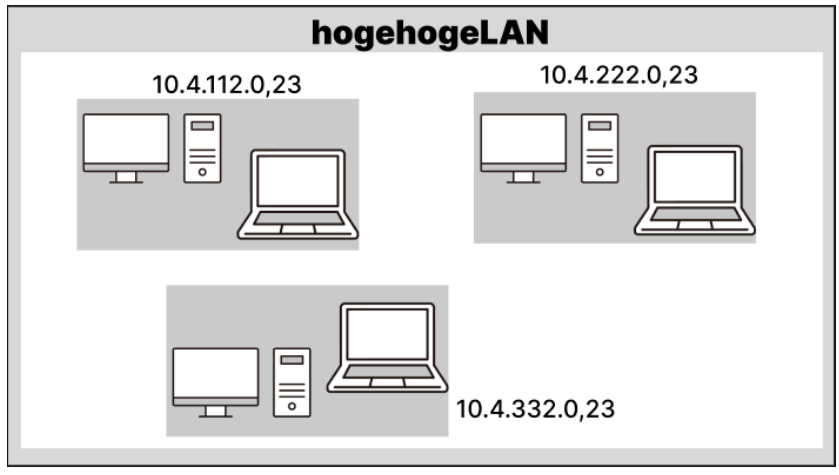


▲ 図 3.1　ICMP のイメージ

3.2 ICMP バージョン

次に ICMP を用いて実装していきます。こちらも ARP と同じように Linux コマンドである ping を実装する過程で改良していきます。ただし、ICMP はルータを超えることができる特性を活用し、3.2 のような、同一 LAN 内で別ネットワークになっている部分に対し、ネットワークごとに接続端末数を取得していきます。元のプログラムは pgDora56 様を参考にさせていただきました [6]。実装する処理の流れは以下の通りです。

1. csv ファイルからネットワーク一覧を取得
2. それぞれのネットワークで扱えるすべての IP を取得
3. ネットワークのすべての IP に対してエコー Type で ICMP プロトコルを実行
4. エコー応答の数を計測
5. 表示



▲ 図 3.2　同一 LAN 内で IP アドレスの違うネットワーク

3.2.1　事前準備

事前に構造体とグローバル変数を定義しておきます。

```
1 type netowork struct {
2       RoomId int
3     Ip     string
4     Mask   int
5   }
6 type Option struct {
7 Address string
8 }
9
10 var count int
```

3.2.2　csv ファイルからネットワーク一覧を取得

今回は対象となるネットワークアドレスを csv ファイルに一覧として用意しておきます。

```
room_id, network, mask
1,10.4.112.0,23
2,10.4.222.0,23
```

```
1 func getIpAndMask() []netowork {
2   file, err := os.Open("./network.csv")
3   if err != nil {
4        log.Fatal(err)
5    }
6    defer file.Close()
7    r := csv.NewReader(file)
8    rows, err := r.ReadAll() // csv を一度に全て読み込む
9    if err != nil {
10    log.Fatal(err)
11    }
12    var net []netowork
13    for i, v := range rows {
14    if i == 0 {
15     continue
16    }
17    id, _ := strconv.Atoi(v[0])
18    mask, _ := strconv.Atoi(v[2])
19    n := netowork{
20        RoomId: id,
21        Ip:     v[1],
22        Mask:   mask,
23    }
24    net = append(net, n)
25    }
26    return net
27  }
```

3.2.3 それぞれのネットワークの IP を取得

ARP の時と同じ処理なので省略します。

3.2.4 ネットワークのすべての IP に対してエコー Type で ICMP プロトコルを実行

エコー Type の番号は 8 番で、ping コマンドで使われているものと同じです。

```
1 func Ping(ip string, m int) int {
2     count = 0
3     addr := &net.IPNet{
4     IP:   net.ParseIP(ip).To4(),
5    Mask: net.CIDRMask(m, 32),
6    }
7
8     out := ips(addr)
9
10     c, err := icmp.ListenPacket("ip4:icmp", "0.0.0.0")
11     if err != nil {
12     panic(err)
13     }
14   defer c.Close()
15
16   for _, v := range out {
17
18     opt := Option{
19     Address: v.String(),
20     }
21     ip, err := net.ResolveIPAddr("ip4", opt.Address)
22     if err != nil {
23      panic(err)
24   }
25
26     go try(c, ip.IP)
27     }
28 }
29
30 func try(c *icmp.PacketConn, ip net.IP) {
31 now := time.Now().UnixMilli()
```

```
32 result := make([]byte, binary.MaxVarintLen64)
33 binary.PutVarint(result, now)
34
35 msg := icmp.Message{
36 Type: ipv4.ICMPTypeEcho,
37 Code: 0,
38 Body: &icmp.Echo{
39 ID:   os.Getpid() & 0xffff,
40 Seq:  1,
41 Data: result,
42 },
43 }
44
45 msgBytes, err := msg.Marshal(nil)
46 if err != nil {
47 return
48 }
49 if _, err := c.WriteTo(
50        msgBytes,
51        &net.IPAddr{IP: ip}
52        ); err != nil {
53 return
54 }
55 }
```

3.2.5　エコー応答の数を計測

```
func Ping(ip string, m int) int {
             . . .
    time.Sleep(time.Second)

    fmt.Println("現在の接続端末数：", count)
    return count
```

```
}

func try(c *icmp.PacketConn, ip net.IP) {
        c.SetDeadline(time.Now().Add(time.Second))

        rb := make([]byte, 1500)
        n, _, _ := c.ReadFrom(rb)
        if err == nil {
                rm, err := icmp.ParseMessage(
                    ipv4.ICMPTypeEcho.Protocol(),
                    rb[:n]
                    )
                if err == nil &&
                rm.Type == ipv4.ICMPTypeEchoReply {
                    echo, ok := rm.Body.(*icmp.Echo)
                    if ok {
                        t, _ := binary.Varint(echo.Data)
                        fmt.Println(
                            ip, ":",
                            time.Now().UnixMicro()-t*1000,
                            " ms"
                            )
                        if time.Now().UnixMilli()-t < 1000 {
                            count++
                        }
                    }
                }
        }
}
```

3.2.6

4 おわりに

参考文献

[1] https://www.cloudflare.com/ja-jp/learning/ddos/glossary/internet-control-message-protocol-icmp

[2] https://www.rfc-editor.org/rfc/rfc792.html

[3] https://www.n-study.com/tcp-ip/arp-format

[4] 福永勇二 (2018)『ネットワークがよくわかる教科書』SB クリエイティブ株式会社発行

[5] https://github.com/google/gopacket/blob/master/examples/arpscan/arpscan.go

[6] https://github.com/pgDora56/PinGo