サイバーセキュリティ

大阪大学大学院 工学研究科
高野 祐輝

目次

1 TCP/IP の基礎

1.1 OSI 参照モデル

インターネットで利用されるプロトコルは、The Internet Engineering Task Force (IETF) という標準化団体により策定され、その標準は Request for Comments (RFC) という名のオープンな仕様として発行されている。例えば、我々が利用しているインターネットプロトコルであるインターネットプロトコル バージョン 4 は、1981 年に 791 番目の RFC として策定された [1]。

IETF 以外の通信に関する標準化団体としては International Telecommunication Union Telecommunication Standardization Sector (ITU-T) や、International Organization for Standardization (ISO) が存在する。実は、1977 年から 1982 年かけて、ITU-T や ISO がコンピュータネットワークの標準通信プロトコルとして、Open Systems Interconnection (OSI) の策定を行っていた。その当時は標準的な通信プロトコルは存在せず、ベンダーごとに様々なプロトコルが利用されていたため、通信プロトコルの統一化が求められていた

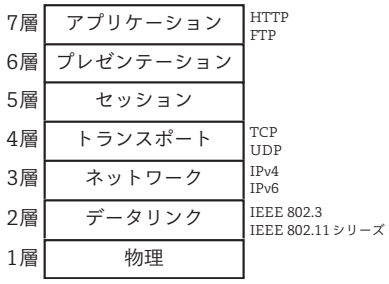

図 1　OSI 参照モデル

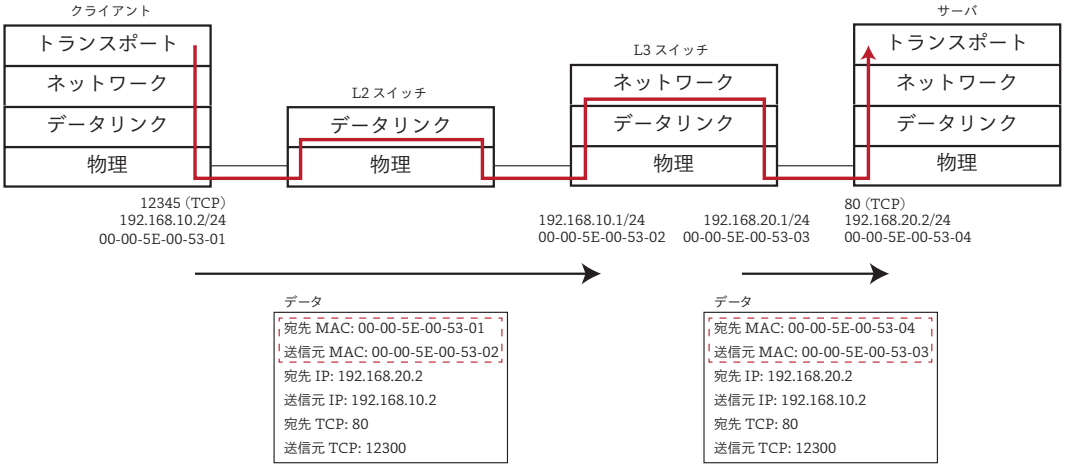

図 2　各層でのデータ転送

のである。しかしながら、最終的に OSI は主流とはならず、IETF によって策定されたインターネットプロトコルが広く利用されるようになっていった。

OSI 自体は残らなかったが、OSI 策定の際に考案された OSI 参照モデルと呼ばれるネットワークの抽象化手法は、今日でも広く受け入れられている。図 1 は、OSI 参照モデルによるネットワークの抽象化モデルを表している。OSI 参照モデルでは、ネットワークの機能を階層構造にもとづいて抽象化しており、この抽象化をレイヤリングなどと呼ぶ。OSI 参照モデルでは、下から順に 1 層に物理層、2 層にデータリンク層、3 層にネットワーク層、4 層にトランスポート層、5 層にセッション層、6 層にプレゼンテーション層、7 層にアプリケーション層が位置する。ちなみに、各層のことをレイヤ 1、レイヤ 2 といったり、更に略して L1、L2 などということもある。

図 2 は各層でデータ転送が行われている様子を示している。[*1] データリンク、ネットワーク、トランスポート層のプロトコルにはそれぞれアドレスがあり、各層は、そのアドレスに基づいて転送を行う。データリンク層プロトコルの一つである IEEE 802 では、アドレスは 42 ビットで表され、16 進数で表現すると 00-00-5E-00-53-02 といった表記になる。図 2 中で宛先 MAC と示される値は、IEEE 802 の宛先 MAC アドレスを示している。なお、MAC は Media Access Control の略である。データリンク層は、ローカルなネットワークでの通信を行うために用いられる。そのため、MAC アドレスはそのローカルな環境では一意に識別

[*1] この図の意味することは現時点では理解できないかもしれないが、この図の意味することろを説明するのが本節の目標であるため、現段階で理解できなくても問題ない。

できる必要がある。データリンク層の詳細については 1.4 節で解説する。

ネットワーク層プロトコルの一つである IP のアドレスは、192.168.10.2/24 という 32 ビットの数値で表され、/24 はネットワークのサブネット長を示している。図 2 では、192.168.10.0/24 と 192.168.20.0/24 というサブネットが示されている。IP は、全世界で通信を行うために用いられるプロトコルであり、基本的には IP アドレスは世界で一意に識別できるように割り当てるのが設計理念となっている（現実的にはそうはなっていないが)。なお、前述のアドレスは IPv4 アドレスであるが、IPv6 の場合は 128 ビットのアドレス空間を持つ。ネットワーク層の詳細については 1.7 節で解説する。

トランスポート層プロトコルの TCP と UDP のアドレスは 16 ビットで示され、一般的にポート番号と呼ばれ、TCP や UDP はポート番号をもとにアプリケーションプロセスの識別を行う。よく利用されるポート番号は、インターネット上で利用される識別情報の管理割当を行っている Internet Assigned Number Authority (IANA) が定義しており [2]、一般的にこのようなポート番号を Well Known ポート番号と呼ぶ。例えば、TCP の 80 番ポートは HTTP で利用され、普段我々が Web を閲覧する際は、Web ブラウザが Web サーバの TCP80 番ポートへ接続する。

図 2 では、クライントからサーバの TCP80 番ポートへむけて通信を行っている様子を示している。一般的に、インターネット上の通信ではデータ中に含まれる各層のアドレスをもとに、L2 または L3 スイッチが転送を行う。L2 スイッチのことをスイッチングハブといったり、L3 スイッチのことをルータということもあるが、本書では L2 スイッチ、L3 スイッチと呼ぶことにする。この図が示すように、L2 スイッチ、L3 スイッチによってデータが転送されても、データ中の IP アドレスとポート番号は変わらないが、MAC アドレスは L3 スイッチでの転送時に更新される。これは、MAC アドレスはローカルなネットワーク内でのみ通用するアドレスであり、L3 スイッチはローカルなネットワーク同士をつなぎ合わせる役割を持っているためである。以降の節では、データリンク、ネットワーク、トランスポートの動きについて詳しく説明する。

重要ポイント

- インターネット関連のプロトコルは、IETF が発行する RFC によって標準化されている
- コンピュータネットワークはレイヤで考えることができる
- Ethernet のアドレスは 48 ビットの MAC アドレス、IPv4 のアドレスは 32 ビットの IPv4 アドレス、IPv6 のアドレスは 128 ビットの IPv6 アドレス、TCP と UDP のアドレスは 16 ビットのポート番号

1.2 おもちゃのネットワークスタック

これより本章では、おもちゃのネットワークスタックを用いて、ネットワークスタックの設計と実装を解説する。おもちゃと言っても、実際に IP ルータや Ethernet ブリッジとして動作するれっきとしたネットワークスタックである。図 3 はおもちゃのネットワークスタックのデータフロー図を示している。この図の下部には、入力と出力用の Ethernet Input/Ouptput Queue というキューがあり、ここで物理的な入出力が行われる。実際に、ネットワークインターフェースカードには入出力用のキューが用意されており、デバイスドライバはこれらキューに対して読み書きすることでデータの送受信を行う。

なお、このおもちゃのネットワークは、Ethernet ブリッジや、IPv4 のルーティングは行うことができるが、TCP のセッション管理などは行えないし、扱えるのは基本的に IP はユニキャストのみで、IP マルチキャスト通信はサポートしていない。また、実際の OS ではネットワークスタックの上にソケットレイヤが配置さ

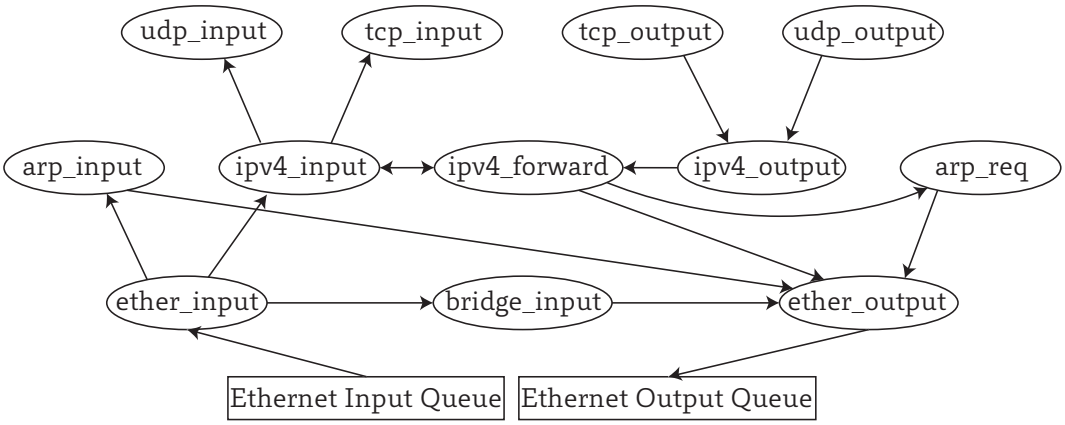

図 3　おもちゃのネットワークスタックのデータフロー図

れ、ネットワークに関する操作が抽象化されているが、おもちゃのネットワークスタックではソケットレイヤは省略されている。すなわち、あくまでも、ネットワークスタックの仕組みから理解してファイアウォールなどを運用するために必要最低限と思われる機能のみが実装されている。

1.3　ネットワークインターフェース

スタックの説明を行う前に、ネットワークインターフェース情報を表すための構造体を説明しよう。ソースコード 1 は、おもちゃのネットワークスタックで定義するネットワークインターフェース用の my_ifnet 構造体となる。

ソースコード 1　ネットワークインターフェースを表す構造体 (my_ifnet.h)

```
1  // インターフェース情報を保持する構造体
2  struct my_ifnet {
3      int idx;                        // インデックス
4      uint8_t ifaddr[6];              // MAC アドレス
5      struct in_addr addr;            // IPv4 アドレス
6      struct in6_addr addr6;          // IPv6 アドレス
7      uint8_t plen;                   // IPv4 プレフィックス長
8      uint8_t plen6;                  // IPv6 プレフィックス長
9      char infile[128];               // 入力UNIX ファイル名
10     char outfile[128];              // 出力UNIX ファイル名
11     int sockfd;                     // 入力先UNIX ドメインソケット
12     struct sockaddr_un outun;       // 出力UNIX アドレス
13     LIST_ENTRY(my_ifnet) pointers; // リスト
14 };
```

my_ifnet 構造体のメンバ変数は基本的にはコメントにあるとおりだが、もう少し詳しく説明したのが表 1 となる。表 1 で示すように、ネットワークインターフェースには各種アドレスが紐付けられる。また、おもちゃのネットワークスタックでは、データの送受信に UNIX ドメインソケットのデータグラム通信を行うため、UNIX ドメインソケット用のデータがいくつが用意されている。

ソースコード 1 はおもちゃのネットワークスタックの割り込みハンドラを表している。割り込みハンドラとは、ネットワークカードにデータが到着した際に呼び出される関数のことを指す。実際の OS では物理的な入力が割り込みハンドラを起動するが、おもちゃのネットワークスタックでは UNIX ドメインソケットへの

表 1　my_ifnet 構造体のメンバ変数

idx	複数インターフェースの番号を識別するためのメンバ変数
addr	インターフェースに対応付けられた IPv4 アドレス
addr6	インターフェースに対応付けられた IPv6 アドレス
plen	IPv4 プレフィックスアドレス（1.7 節にて解説）
plen6	IPv6 プレフィックスアドレス（1.7 節にて解説）
infile	データ受信を行うための UNIX ドメインソケットへのファイル名
outfile	データ送信を行うための UNIX ドメインソケットへのファイル名
sockfd	データ受信用の UNIX ドメインソケットへのデスクリプタ
outun	データ送信用の UNIX ドメインソケットへのアドレス
pointers	複数インターフェースをリストで管理するためのポインタ。sys/queue.h を利用

入力があったときに dev_input 関数を呼び出すようにしている。

ソースコード 2　割り込みハンドラ (my_ifnet.c)

```
1  /*
2   * インターフェース入力割り込み関数
3   * 引数:
4   *   fd: UNIX ドメインソケットへのファイルデスクリプタ
5   */
6  void dev_input(int fd) {
7      for (struct my_ifnet *np = LIST_FIRST(&ifs); np != NULL;
8           np = LIST_NEXT(np, pointers)) {
9          if (np->sockfd == fd) {
10             char buf[4096];
11             ssize_t size;
12         again:
13             size = recv(fd, buf, sizeof(buf), 0);
14             if (size < 0) {
15                 if ((errno) == EAGAIN)
16                     goto again;
17
18                 perror("recv");
19                 break;
20             }
21
22             ether_input(np, (struct ether_header *)buf, size);
23             break;
24         }
25     }
26 }
```

ソースコード 1 では引数に入力用の UNIX ドメインソケットを受け取り、7〜8 行目で対応する UNIX ドメインソケットを持つ my_ifnet 構造体をリストから検索している。入力インターフェースの my_ifnet 構造体が見つかったら（9 行目）、13 行目でデータ読み込みを行っている。14〜20 行目エラー処理で、読み込みに失敗した場合は errno が EAGAIN であれば、再度読み直しそれ以外であれば読み込み失敗として関数を抜ける。データを読み込んだ後、22 行目で ether_input 関数を呼び出して実際の処理に入る。これは図 3 で示される、Ethernet Input Queue から ether_input 関数へのデータフローに相当する。

重要ポイント

- ネットワークインターフェースカードにデータが到着した際に、OS で設定した割り込みハンドラと呼ばれる関数が呼び出される
- 割り込みハンドラから、実際にネットワーク処理を行うための関数が呼ばれる

1.4 データリンク層

ソースコード 3 は Ethernet (IEEE 802.3) プロトコルのヘッダ構造体を示している。我々が普段利用している無線や有線の Ethernet では、内部的にはこのようなフォーマットのヘッダがデータの先頭に付与され、その後に IP ヘッダ、TCP ヘッダなどのより上位のヘッダが続き、最後にアプリケーションデータが続く。もう少し正確に言うと、ソースコード 3 で示す Ethernet ヘッダの前にプリアンブルなどのハードウェアで利用されるデータが続くが、本書ではその説明は割愛する。

ソースコード 3　Ethernet プロトコルヘッダ定義 (/usr/include/net/ethernet.h)

```
1  #define ETHER_ADDR_LEN  6       /* Ethernet address length             */
2
3  /*
4   * The length of the combined header.
5   */
6  struct  ether_header {
7          u_int8_t  ether_dhost[ETHER_ADDR_LEN];
8          u_int8_t  ether_shost[ETHER_ADDR_LEN];
9          u_int16_t ether_type;
10 };
11
12 #define ETHERTYPE_IP            0x0800  /* IP protocol */
13 #define ETHERTYPE_ARP           0x0806  /* Addr. resolution protocol */
14 #define ETHERTYPE_IPV6          0x86dd  /* IPv6 */
```

ソースコード 3 で示されるように、Ethernet ヘッダの構造体は、OpenBSD では、/usr/include/net/ethernet.h にて定義されている。なお、以降特に断りがない限り対象とする OS は OpenBSD とし、/usr/include のパスから始まるソースコードは、OS が提供するソースコードであるとする。1 行目の ETHER_ADDR_LEN では、Ethernet アドレス（MAC アドレス）のバイト数を 6 バイトと定義している。6 行目以降が Ethernet ヘッダを示す ether_header 構造体となる。ether_header 構造体では、ether_dhost と ether_shost というメンバ変数を持ち、それぞれ宛先 MAC アドレスと送信元 MAC アドレスを示している。ether_type メンバ変数は、Ethernet ヘッダ以降に続くプロトコル種類を示している。

ether_type メンバ変数で利用できる値は IANA によって定義されている [3]。例えば、IPv4 が続く場合は 16 進数表記で 0x0800 という値が ether_type に格納される。他には、IPv6 の場合は 0x08DD、仮想的な LAN を構築するための IEEE 802.1Q VLAN プロトコルの場合は 0x8100 が格納される。この値は ether_header 構造体と同じファイルにて定義されており、ソースコード 3 に 12〜14 行目に一部抜粋してある。ただし、ether_type メンバ変数のバイトオーダはビッグエンディアンであるため、比較や格納する際はバイトオーダを変換してから行わなければならない。

ソースコード 4 はおもちゃのネットワークスタックの Ethernet フレームを受け取り処理を行う ether-net_input 関数である。この関数では、引数に入力インターフェースを指す my_ifnet 構造体のポインタ、入力 Ethernet フレームへのポインタ、フレーム長をとり、Ethernet ヘッダ中のプロトコルタイプに応じて上位レ

イヤの関数に渡している。

ソースコード 4　ether_input 関数 (ether.c)

```
1  // ADDR が IPv4 ブロードキャストアドレスなら真、それ以外なら偽を返すマクロ
2  #define IS_BROADCAST(ADDR)                                                    \
3      (((ADDR)[0] == 0xFF) && ((ADDR)[1] == 0xFF) && ((ADDR)[2] == 0xFF) &&     \
4       ((ADDR)[3] == 0xFF) && ((ADDR)[4] == 0xFF) && ((ADDR)[5] == 0xFF))
5
6  /*
7   * Ethernet フレーム入力関数
8   * 引数:
9   *   ifp: 入力インターフェース
10  *   eh: 入力フレーム
11  *   len: 入力フレーム長
12  */
13 void ether_input(struct my_ifnet *ifp, struct ether_header *eh, int len) {
14     printf("ether_input:\n");
15     printf("␣␣␣␣IF#:␣%d\n", ifp->idx);
16     printf("␣␣␣␣SRC␣MAC:␣%02X-%02X-%02X-%02X-%02X-%02X\n", eh->ether_shost[0],
17            eh->ether_shost[1], eh->ether_shost[2], eh->ether_shost[3],
18            eh->ether_shost[4], eh->ether_shost[5]);
19     printf("␣␣␣␣DST␣MAC:␣%02X-%02X-%02X-%02X-%02X-%02X\n", eh->ether_dhost[0],
20            eh->ether_dhost[1], eh->ether_dhost[2], eh->ether_dhost[3],
21            eh->ether_dhost[4], eh->ether_dhost[5]);
22     printf("\n");
23
24     if (IS_BROADCAST(eh->ether_dhost)) {
25         // ブロードキャストアドレスの場合、ブリッジ処理へ
26         if (IS_L2BRIDGE)
27             bridge_input(ifp, eh, len);
28     } else if (memcmp(ifp->ifaddr, eh->ether_dhost, ETHER_ADDR_LEN) != 0) {
29         // 宛先MAC アドレスが自インターフェース宛でないならブリッジ処理を行い終了
30         if (IS_L2BRIDGE)
31             bridge_input(ifp, eh, len);
32         return;
33     }
34
35     switch (ntohs(eh->ether_type)) {
36     case ETHERTYPE_IP: // IPv4 入力
37         ipv4_input((struct ip *)((uint8_t *)eh + ETHER_HDR_LEN));
38         break;
39     case ETHERTYPE_IPV6: // IPv6 入力
40         ipv6_input((struct ip6_hdr *)((uint8_t *)eh + ETHER_HDR_LEN));
41         break;
42     case ETHERTYPE_ARP: // ARP 入力
43         arp_input(ifp, (struct arphdr *)((uint8_t *)eh + ETHER_HDR_LEN));
44         break;
45     default:
46         printf("eh->ether_type␣is␣neither␣IPv6␣nor␣IPv6\n");
47         return;
48     }
49
50     return;
51 }
```

14〜22 行目では入力 Ethernet フレームの送信元と宛先 MAC アドレスを表示している。24 行目では、宛先がブロードキャストアドレスかチェックしている。すなわち、FF-FF-FF-FF-FF-FF という MAC アドレ

スが Ethernet のブロードキャストアドレスであるため、この値かどうかを、IS_BROADCAST マクロで判定している。宛先がブロードキャストアドレスの場合かつ、L2 ブリッジが有効であるなら、L2 ブリッジ処理を行う bridge_input 処理を行い、自身のネットワークスタック入力処理へと進む。28 行目では、宛先が受信したネットワークインターフェースの MAC アドレスと同じであるか（すなわち自分宛てであるか）をチェックし、自分宛てで無いならば、L2 ブリッジが有効の場合に L2 ブリッジ処理を行う関数へデータを渡し、ether_input 処理を終了する。L2 ブリッジが有効かどうかは、IS_L2BRIDGE というマクロで判定する。

35 行目から始まる switch 文では、上位のレイヤのプロトコルタイプを判別して、対応するプロトコルの関数に渡している。おもちゃのネットワークスタックでは、IPv4 のみに対応しているが、例のために IPv6 用のダミー関数も用意している。また、IPv4 で通信を行うためには、アドレス解決プロトコル（Address Resolution Protocol、ARP） [4] というプロトコルで MAC アドレスと IPv4 アドレスの対応の解決を行わなければならない。そのため、おもちゃのネットワークスタックでも ARP をサポートしている。

なお、ETHERTYPE_IP、ETHERTYPE_IPV6、ETHERTYPE_ARP といった定義は、ソースコード 3 で示したように、/usr/include/net/ethernet.h で定義されている。35 行目では ntohs という関数を利用するが、これは 2 バイト変数のバイトオーダをホストバイトオーダ（ホスト CPU に依存）からからネットワークバイトオーダ（ビッグエンディアン）に変換する標準 C ライブラリ関数となる。

重要ポイント

- 入力インターフェースの MAC アドレスと、Ethernet ヘッダ中の宛先 MAC アドレスを比較して、自身宛ての Ethernet フレームか判別する
- ブロードキャスト MAC アドレス（FF-FF-FF-FF-FF-FF）の場合は自身宛てと判別する
- Ethernet ヘッダ中のプロトコルタイプフィールドを判別して、IPv4、IPv6、ARP など上位層のプロトコル種別を判別する

1.5 L2 ブリッジ

図 4 は L2 ブリッジの動作を示した図となる。L2 ブリッジにはポートと呼ばれる物理的 [*2]なポートと呼ばれる接続口があり、それぞれのポートと各ホストが接続されている。また、一般的に L2 ブリッジは MAC アドレステーブルと呼ばれるテーブルを保持しており、この MAC アドレステーブルは MAC アドレスとポート番号を保持している。

図 4 では、4 つのノード A〜D が、1 つの L2 ブリッジに接続されており、初期状態では L2 ブリッジの MAC アドレステーブルは空の状態である。図の上部ではホスト A からホスト C の MAC アドレス 00-00-5E-00-53-03 へ向けて Ethernet フレームが送信され、この Ethernet フレームは L2 ブリッジにより全てのホストへ向けて転送されている。これは、L2 ブリッジが Ethernet フレームを転送する際には、転送すべきポートを MAC アドレステーブルから参照して決定するが、MAC アドレステーブルに MAC アドレスが存在しない場合は、全てのポートへ転送するためである。

L2 ブリッジは、一度 Ethernet フレームを転送すると Ethernet フレームの送信元 MAC アドレスとポート番号を記憶する。これを示したのが図 4 の下部の MAC アドレステーブルとなる。本図の下部では、Ethernet フレームを受け取ったホスト C が、ホスト A の MAC アドレスである 00-00-5E-00-53-01 に対して何かしら応答している。しかし、ここでは L2 ブリッジは全てのホストへ向けて Ethernet フレームを転送するのでは

[*2] 仮想的な場合もある

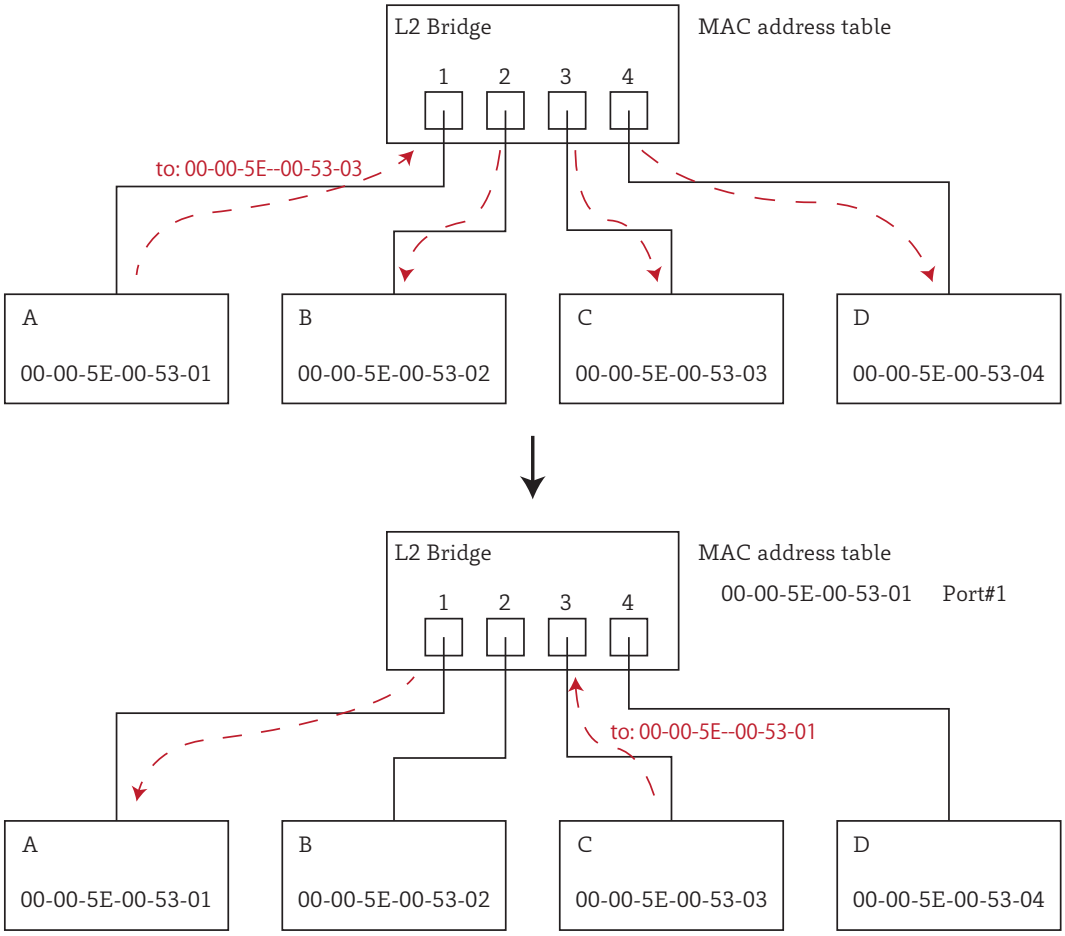

図 4　L2 ブリッジの動作図

なく、ホスト A のみに対して転送されている。これは、MAC アドレステーブルにすでにホスト A の MAC アドレスとポート番号が保存されているためである。

ソースコード 5 は、おもちゃのネットワークでの MAC アドレステーブルの実装を示している。

ソースコード 5　MAC アドレステーブルの実装

```
1  // MACアドレスのハッシュ値を計算するマクロ
2  #define MACHASH(ADDR)                                                \
3      ((ADDR)[0] ^ (ADDR)[1] ^ (ADDR)[2] ^ (ADDR)[3] ^ (ADDR)[4] ^ (ADDR)[5])
4
5  // MACアドレステーブル定義
6  struct mac2if {
7      uint8_t addr[ETHER_ADDR_LEN];
8      struct my_ifnet *ifp;
9  };
10
11 struct mac2if mactable[256]; // MACアドレステーブルのインスタンス
12
13 /*
14  * MACアドレステーブルにMACアドレスを追加する関数
15  * 引数:
16  *   ifp: 入力インターフェース
```

```
17  *   eh: 入力フレーム
18  */
19 static void add2mactable(struct my_ifnet *ifp, struct ether_header *eh) {
20     // 送信元MACアドレスがブロードキャストアドレスならテーブルに追加しない
21     if (IS_BROADCAST(eh->ether_shost))
22         return;
23
24     // 送信元MACアドレスからハッシュ値を計算
25     uint8_t hash = MACHASH(eh->ether_shost);
26
27     // テーブルに追加
28     mactable[hash].ifp = ifp;
29     memcpy(mactable[hash].addr, eh->ether_shost, ETHER_ADDR_LEN);
30 }
31
32 /*
33  * MACアドレステーブルから対応するインターフェースを取得する関数
34  * 返り値:
35  *   対応するmy_ifnet構造体へのポインタ。
36  *   ただし､テーブルにMACアドレスが存在しない場合はNULLが返る。
37  * 引数:
38  *   eh: 入力フレーム
39  */
40 static struct my_ifnet *find_interface(struct ether_header *eh) {
41     // 宛先MACアドレスからハッシュ値を計算
42     uint8_t hash = MACHASH(eh->ether_dhost);
43
44     // 送信元MACアドレスとMACアドレステーブルのMACアドレスとを比較
45     if (memcmp(mactable[hash].addr, eh->ether_dhost, ETHER_ADDR_LEN) != 0)
46         return NULL;
47
48     return mactable[hash].ifp;
49 }
```

MAC アドレステーブルは 256 個の配列から構成され、配列のインデックスは MAC アドレスの値を排他的論理和で計算する。ソースコード 5 の 2 行目は、MAC アドレスのハッシュ値を計算する関数で、5〜11 行目は MAC アドレステーブルの定義となる。mac2if 構造体をみてわかるように、MAC アドレステーブルを利用すると MAC アドレスからインターフェースを得ることが出来る。19〜30 行目で、MAC アドレステーブルへ新規 MAC アドレスを追加する add2mactable 関数が定義されている。また、40〜49 行目では、MAC アドレステーブルから、MAC アドレスをキーとしてインターフェースを取得する find_interface 関数を定義している。これらは、11 行目で定義した配列に値を格納、取得する単純な関数であるので動作は各自で確認してほしい。

ソースコード 6 は実際にブリッジ処理を行う bridge_input 関数となる。この関数では、引数に入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、入力 Ethernet フレームをさす ether_header 構造体へのポインタ変数である eh と、フレーム長をさす len をとる。

ソースコード 6　bridge_input 関数

```
1  /*
2   * ブリッジ処理を行う関数
3   * 引数:
4   *   ifp: 入力インターフェース
5   *   eh: 入力フレーム
```

```
6  *   len: 入力フレーム長
7  */
8  static void bridge_input(struct my_ifnet *ifp, struct ether_header *eh,
9                           int len) {
10     // MACアドレステーブルを検索
11     struct my_ifnet *outif = find_interface(eh);
12
13     if (outif) {
14         // MACアドレステーブルにキャッシュされていた場合、そのインターフェースへ送信
15         ether_output(outif, eh, len);
16     } else {
17         // MACアドレステーブルにない場合、受信インターフェース以外の全てのインターフェースへ送信
18         for (struct my_ifnet *np = LIST_FIRST(&ifs); np != NULL;
19              np = LIST_NEXT(np, pointers)) {
20             if (ifp != np)
21                 ether_output(np, eh, len);
22         }
23     }
24
25     // MACアドレステーブルへキャッシュ
26     add2mactable(ifp, eh);
27 }
```

この関数では、まず、11 行目で、MAC アドレステーブルを検索し出力先インターフェースを取得する。その後、出力先インターフェースが取得できれば、15 行目でそのインターフェースへ出力し、見つからなければ 18〜21 行目で全てのインターフェースへ出力する。そして最後に、26 行目で、受信した Ethernet フレームの送信元 MAC アドレスを MAC アドレステーブルへ追加する。

本設では、内部に MAC アドレステーブルを持つ L2 ブリッジについて説明したが、MAC アドレステーブルを持たず、全てのインターフェースへと転送するものもある。一般的に、このような L2 ブリッジはリピータハブとも呼ばれたり、バカハブ（馬鹿ハブ）とも呼んだりする。スイッチングハブ、スマートスイッチ、インテリジェントスイッチなどと呼ぶときは全て、MAC アドレステーブルを内部的に持つが、バカハブ、リピータハブと呼ぶときには内部的に MAC アドレステーブルは持っていない。

重要ポイント

- スイッチングハブは内部にフレーム転送用の MAC アドレステーブルを持つ
- Ethernet フレームの転送は、MAC アドレステーブルの情報を用いて行われる
- 転送すべき Ethernet フレームの宛先 MAC アドレスが MAC アドレステーブルにない場合は、全てのインターフェースへと転送する
- Ethernet フレームが入力されたタイミングで MAC アドレステーブルが更新される

1.6 アドレス解決プロトコル (ARP)

Ethernet での通信は MAC アドレスベースで行われる、より上位層の IP は MAC アドレスではなく IP アドレスで通信を行う。したがって、IP パケットを Ethernet フレームで転送するためには、どの MAC アドレスがどの IP アドレスに対応しているかを知らなければならない。ARP は、IPv4 アドレスから MAC アドレスの対応を得るために使うプロトコルである。

図 5 は ARP の動作を示した図となる。この図では 3 つのホスト A、B、C があり、A が B へと通信するた

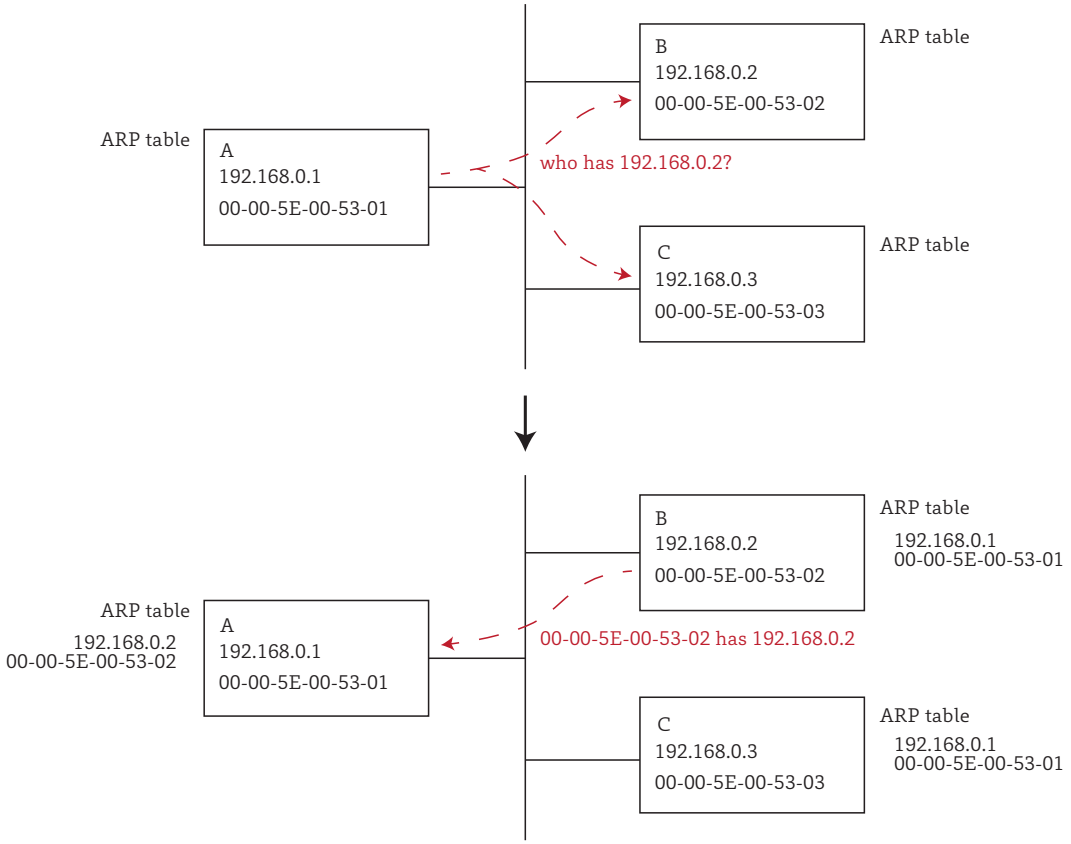

図 5　ARP の動作図

めに ARP で IPv4 アドレスと MAC アドレスの対応の解決を行っている。それぞれのホストは ARP テーブルと呼ばれる、IPv4 アドレスと MAC アドレスの対応を保存しておくテーブルを持っており、初期状態では ARP テーブルは空である。

ホスト A がホスト B に通信を行うために、ホスト A は、まず ARP リクエストと呼ばれるパケットを Ethernet ブロードキャストでリンク内の全てのノードに送信する（ただしここで、ホスト A はホスト B の IPv4 アドレスを知っているとする）。これは図 5 の上部に赤の破線で示されるパケットであり、この ARP リクエストには、192.168.0.2 という IPv4 アドレスを持っているノードは誰かを問い合わせている。ARP リクエストを受け取ったホスト B、C は自身の ARP テーブルにホスト A の IPv4 アドレスと MAC アドレスを記憶しておく。すると、図 5 の下部に示されるように、ホスト B、C の ARP テーブルが更新される。

ARP リクエストを受け取ったホスト B は、リクエスト先の IPv4 アドレスが自身のアドレスと一致するため、ARP リプライをホスト A に送信する。ARP リプライを受け取ったホスト A は、自身の ARP テーブルにホスト B の情報を追加して、その後、実際に IPv4 アドレスでの通信を開始する。

ARP ヘッダ構造体と Ethernet で用いる ARP 構造体は、ソースコード 7 と 8 に示すように、/usr/include/net/if_arp.h と/usr/include/netinet/if_ether.h にて定義されている。

ソースコード 7　ARP 構造体 (/usr/include/net/if_arp.h)

```
1 /*
2  * Address Resolution Protocol.
3  *
```

```
 * See RFC 826 for protocol description.  ARP packets are variable
 * in size; the arphdr structure defines the fixed-length portion.
 * Protocol type values are the same as those for 10 Mb/s Ethernet.
 * It is followed by the variable-sized fields ar_sha, arp_spa,
 * arp_tha and arp_tpa in that order, according to the lengths
 * specified.  Field names used correspond to RFC 826.
 */
struct  arphdr {
        u_int16_t ar_hrd;       /* format of hardware address */
#define ARPHRD_ETHER    1       /* ethernet hardware format */
#define ARPHRD_IEEE802  6       /* IEEE 802 hardware format */
#define ARPHRD_FRELAY   15      /* frame relay hardware format */
#define ARPHRD_IEEE1394 24      /* IEEE 1394 (FireWire) hardware format */
        u_int16_t ar_pro;       /* format of protocol address */
        u_int8_t  ar_hln;       /* length of hardware address */
        u_int8_t  ar_pln;       /* length of protocol address */
        u_int16_t ar_op;        /* one of: */
#define ARPOP_REQUEST   1       /* request to resolve address */
#define ARPOP_REPLY     2       /* response to previous request */
#define ARPOP_REVREQUEST 3      /* request protocol address given hardware */
#define ARPOP_REVREPLY  4       /* response giving protocol address */
#define ARPOP_INVREQUEST 8      /* request to identify peer */
#define ARPOP_INVREPLY  9       /* response identifying peer */
/*
 * The remaining fields are variable in size,
 * according to the sizes above.
 */
#ifdef COMMENT_ONLY
        u_int8_t  ar_sha[];     /* sender hardware address */
        u_int8_t  ar_spa[];     /* sender protocol address */
        u_int8_t  ar_tha[];     /* target hardware address */
        u_int8_t  ar_tpa[];     /* target protocol address */
#endif
};
```

ソースコード 8　Ethernet 用 ARP 構造体 (/usr/include/netinet/if_ether.h)

```
/*
 * Ethernet Address Resolution Protocol.
 *
 * See RFC 826 for protocol description.  Structure below is adapted
 * to resolving internet addresses.  Field names used correspond to
 * RFC 826.
 */
struct  ether_arp {
        struct  arphdr ea_hdr;  /* fixed-size header */
        u_char  arp_sha[ETHER_ADDR_LEN];        /* sender hardware address */
        u_char  arp_spa[4];     /* sender protocol address */
        u_char  arp_tha[ETHER_ADDR_LEN];        /* target hardware address */
        u_char  arp_tpa[4];     /* target protocol address */
}
```

表 2 は arphdr 構造体のメンバ変数を説明した表となる。基本的に、ar_op 変数以外の値は Ethernet と IPv4 を扱うときは表で示した値で固定となる。ただし、構造体定義からもわかるように、ARP は Ethernet や IPv4 に限らず様々なプロトコルで利用可能な設計となっている。ether_arp 構造体のメンバ変数は、MAC アドレスと IPv4 アドレスを保存する変数であることは自明のため詳細は割愛する。

表 2　arphdr 構造体のメンバ変数（全てビッグエンディアン）

ar_hrd	L2 プロトコル識別子。Ethernet の場合は 1
ar_pro	L3 プロトコル識別子。IPv4 の場合は ETHERTYPE_IP
ar_hln	L2 アドレスのバイト数。MAC アドレスの場合は 6 バイト
ar_pln	L3 アドレスのバイト数。IPv4 アドレスの場合は 4 バイト
ar_op	ARP の種類。リクエストの場合は 1 で、リプライの場合は 2

次に、実際におもちゃのネットワークスタックで ARP 処理を行う関数を見ていく。ソースコード 9 は、ARP パケットを受け取り、リクエストやリプライなどそれぞれに対応した関数を呼び出す arp_input 関数である。

ソースコード 9　arp_input 関数

```
1  /*
2   * ARP リクエスト及び応答を受け取る関数
3   * 引数:
4   *   ifp: 入力インターフェース
5   *   arph: 入力ARP
6   */
7  void arp_input(struct my_ifnet *ifp, struct arphdr *arph) {
8      // Ethernet 以外は未対応
9      if (ntohs(arph->ar_hrd) != ARPHRD_ETHER || arph->ar_hln != ETHER_ADDR_LEN)
10         return;
11
12     // IP 以外は未対応
13     if (ntohs(arph->ar_pro) != ETHERTYPE_IP ||
14         arph->ar_pln != sizeof(struct in_addr))
15         return;
16
17     switch (ntohs(arph->ar_op)) {
18     case ARPOP_REQUEST:
19         // ARP リクエストを受け取り応答
20         arp_req_input(ifp, arph);
21         return;
22     case ARPOP_REPLY:
23         // リプライを受け取って送信バッファ中のフレームを送信
24         arp_reply_input(ifp, arph);
25         return;
26     default:
27         // ARP リクエストとリプライ以外は未対応
28         return;
29     }
30 }
```

arp_input 関数は引数に入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、入力 ARP パケットをさす arphdr 構造体へのポインタ変数である arph をとる。9、10 行目で L2 プロトコルと L2 プロトコルアドレスの長さをチェックし、Ethernet かつ 6 バイト意外であるなら、未対応として処理を終了する。13〜15 行目では L3 プロトコルが IPv4 であるかをチェックし、そうでないなら処理を終了する。17 行目で ARP プロトコルの種類を判別し、ARP リクエストであれば 20 行目で arp_req_input 関数を呼び出し、ARP リプライであれば arp_reply_input 関数を呼び出す。ただし、おもちゃのネットワークスタックでは ARP リクエストとリプライ以外は未対応であるため、これら以外の ARP パケットが来た場合は何もせず

に処理を終了する。

ソースコード 10 は ARP リクエストを送信するための arp_req 関数である。図 5 上部のホスト A は、この arp_req 関数を用いて ARP リクエストを送信する。

ソースコード 10　arp_req 関数

```
1  /*
2   * ARP リクエストを送信
3   * 引数:
4   *   ifp: 送信を行うインターフェース
5   *   addr: 問い合わせを行うIP アドレスへのポインタ
6   */
7  void arp_req(struct my_ifnet *ifp, struct in_addr *addr) {
8      uint8_t buf[ETHER_HDR_LEN + sizeof(struct ether_arp)];
9      struct ether_header *eh = (struct ether_header *)buf;
10     struct ether_arp *req = (struct ether_arp *)(buf + ETHER_HDR_LEN);
11
12     // Ethener ヘッダ設定
13     memcpy(eh->ether_shost, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
14     memset(eh->ether_dhost, 0xff, ETHER_ADDR_LEN); // 宛先MAC はブロードキャスト
15     eh->ether_type = htons(ETHERTYPE_ARP);
16
17     // ARP ヘッダ設定
18     req->ea_hdr.ar_hrd = ntohs(ARPHRD_ETHER); // ハードウェアタイプ (Ethernet)
19     req->ea_hdr.ar_pro = ntohs(ETHERTYPE_IP); // プロトコルタイプ (IP)
20     req->ea_hdr.ar_hln = ETHER_ADDR_LEN; // 6バイト。MAC アドレスサイズ
21     req->ea_hdr.ar_pln = sizeof(struct in_addr); // 4バイト。IPv4 アドレスサイズ
22     req->ea_hdr.ar_op = ntohs(ARPOP_REQUEST); // ARP リクエスト
23
24     // ARP リクエスト設定
25     memcpy(req->arp_sha, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
26     memcpy(req->arp_spa, &ifp->addr, sizeof(req->arp_spa)); // 送信元IP は自分
27     memset(req->arp_tha, 0xff, ETHER_ADDR_LEN); // 宛先MAC はブロードキャスト
28     memcpy(req->arp_tpa, addr, sizeof(*addr)); // 問い合わせIP
29
30     ether_output(ifp, eh, ETHER_HDR_LEN + sizeof(struct ether_arp));
31 }
```

arp_req 関数は出力先インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、問い合わせ IPv4 アドレスをさす in_addr 構造体へのポインタ変数である addr を引数にとる。8 行目のでは送信 ARP パケット用のバッファを作成しており、9、10 行目で、Ethernet ヘッダと ARP のためのデータを格納する先のアドレスを計算している。13、14 行目では Ethernet ヘッダ情報を設定しており、送信元アドレスは出力先インターフェースの MAC アドレスとしている。その一方、宛先アドレスは、ネットワーク全体に届くようブロードキャストアドレス（FF-FF-FF-FF-FF-FF）を設定している。また、プロトコルタイプは今回は ARP であるため、ETHERTYPE_ARP を設定している。18〜22 行目では ARP ヘッダの設定をしており、これは前述したとおりである。25〜28 行目では ARP リクエストパケットに IPv4 アドレスと MAC アドレスを設定している。送信元の MAC と IPv4 アドレスは出力先インターフェースのアドレスであり、ターゲットの IPv4 アドレスは問い合わせ IPv4 アドレスである。ただし、問い合わせ MAC アドレスは不明なので、ここではブロードキャストに設定している。ARP パケットに値を設定した後、最後の 30 行目で ethernet_output 関数を呼び出し、ARP パケットを送信する。

ソースコード 11 は ARP リクエストを受け取り ARP リプライを返信する arp_req_input 関数となる。

ソースコード 11　arp_req_input 関数

```
1  /*
2   * ARP リクエストを受け取り、ARP リプライを返す関数
3   * 引数:
4   *   ifp: 受信したインターフェース
5   *   arph: ARP リクエストへのポインタ
6   */
7  static void arp_req_input(struct my_ifnet *ifp, struct arphdr *arph) {
8      struct ether_arp *req = (struct ether_arp *)arph;
9      uint8_t buf[ETHER_HDR_LEN + sizeof(struct ether_arp)];
10     struct ether_header *eh = (struct ether_header *)buf;
11     struct ether_arp *reply = (struct ether_arp *)(buf + ETHER_HDR_LEN);
12
13     char addr[16];
14     inet_ntop(PF_INET, req->arp_tpa, addr, sizeof(addr));
15     printf("ARP: who has %s?\n", addr);
16
17     // ARP テーブルに追加
18     add2arptable((struct in_addr *)req->arp_spa, req->arp_sha);
19
20     // 問い合わせIPv4 アドレスが自身の IPv4 アドレスかチェック
21     struct in_addr *tpa = (struct in_addr *)req->arp_tpa;
22     if (tpa->s_addr != ifp->addr.s_addr)
23         return;
24
25     // Ethener ヘッダ設定
26     memcpy(eh->ether_shost, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
27     memcpy(eh->ether_dhost, req->arp_sha, ETHER_ADDR_LEN); // 宛先MAC
28     eh->ether_type = htons(ETHERTYPE_ARP);
29
30     // ARP ヘッダ設定
31     reply->ea_hdr.ar_hrd = ntohs(ARPHRD_ETHER); // ハードウェアタイプ (Ethernet)
32     reply->ea_hdr.ar_pro = ntohs(ETHERTYPE_IP); // プロトコルタイプ (IP)
33     reply->ea_hdr.ar_hln = ETHER_ADDR_LEN; // 6バイト。MAC アドレスサイズ
34     reply->ea_hdr.ar_pln = sizeof(struct in_addr); // 4バイト。IPv4 アドレスサイズ
35     reply->ea_hdr.ar_op = ntohs(ARPOP_REPLY); // ARP リプライ
36
37     // ARP リプライ設定
38     memcpy(reply->arp_sha, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
39     memcpy(reply->arp_spa, &ifp->addr, sizeof(reply->arp_spa)); // 送信元IP は自分
40     memcpy(reply->arp_tha, req->arp_sha, ETHER_ADDR_LEN);          // 宛先MAC
41     memcpy(reply->arp_tpa, req->arp_spa, sizeof(reply->arp_tpa)); // 質問先IP
42
43     ether_output(ifp, eh, ETHER_HDR_LEN + sizeof(struct ether_arp));
44 }
```

arp_req_input 関数は引数に、入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、ARP リクエストへのポインタ arph をとる。8 行目では、arph ポインタを ether_arp 構造体へのポインタにキャストしている。9〜11 行目では、ARP リプライ用のバッファを確保し、そこから Ethernet ヘッダと ARP リプライ構造体へのアドレスを計算している。13〜15 行目は ARP リクエストの内容を表示しているのみである。18 行目は ARP リクエストの情報を元に ARP テーブルにデータを追加している。これは図 5 上部でホスト B と C が ARP リクエストを受け取り、下部でホスト B と C が自身の ARP テーブルに情報を追加している動作に相当する。21〜23 行目では、ARP リクエストの問い合わせ IPv4 アドレスが受信したインターフェースの IPv4 アドレスかをチェックし、自身宛でないなら処理を終了する。26〜28 行目では

Ethernet ヘッダの設定、31〜35 行目では ARP ヘッダの設定、38〜41 行目では ARP リプライの設定を行っており、最後に 43 行目で ether_output 関数を呼び出て ARP リプライを送信する。

ソースコード 12 は ARP リプライを受け取り、送信バッファで待機中のフレームを送信する関数となる。この送信バッファは、はじめに IPv4 パケットを送信する場合、ARP テーブルに対応する IPv4 パケットがないため必要となる。対応する ARP エントリがない場合、一旦送信バッファに退避してから ARP リクエストを送信して、アドレス解決を行った後に退避しておいた IPv4 パケットを実際に送信する。

ソースコード 12　arp_reply_input 関数

```
1  /*
2   * ARP リプライを受け取り、送信バッファ中のフレームを送信する関数
3   * 引数:
4   *   ifp: 入力インターフェース
5   *   arph:
6   */
7  static void arp_reply_input(struct my_ifnet *ifp, struct arphdr *arph) {
8      struct ether_arp *rep = (struct ether_arp *)arph;
9
10     char addr[16];
11     inet_ntop(PF_INET, rep->arp_spa, addr, sizeof(addr));
12     printf("ARP:␣%02X-%02X-%02X-%02X-%02X-%02X␣has␣%s\n", rep->arp_sha[0],
13            rep->arp_sha[1], rep->arp_sha[2], rep->arp_sha[3], rep->arp_sha[4],
14            rep->arp_sha[5], addr);
15
16     // ARP テーブルに追加
17     add2arptable((struct in_addr *)rep->arp_spa, rep->arp_sha);
18
19     // 送信バッファ中のフレームを送信
20     struct sendbuf *np;
21     for (np = sbuf.lh_first; np != NULL;) {
22         if (np->eh->ether_type == htons(ETHERTYPE_IP)) {
23             // ARP テーブルに宛先 IPv4 アドレスがあるか検索
24             struct ip2mac *mac = find_mac(&np->nextip);
25             if (mac == NULL) {
26                 np = np->pointers.le_next;
27                 continue;
28             }
29
30             // 宛先MAC アドレスを設定
31             memcpy(np->eh->ether_dhost, mac->macaddr, ETHER_ADDR_LEN);
32
33             // インターフェースへ出力
34             ether_output(ifp, np->eh, np->ethlen);
35
36             // バッファを解放
37             free(np->eh);
38             struct sendbuf *tmp = np->pointers.le_next;
39             LIST_REMOVE(np, pointers);
40             free(np);
41             np = tmp;
42         }
43     }
44 }
```

引数はこれまでと同じく、入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、ARP

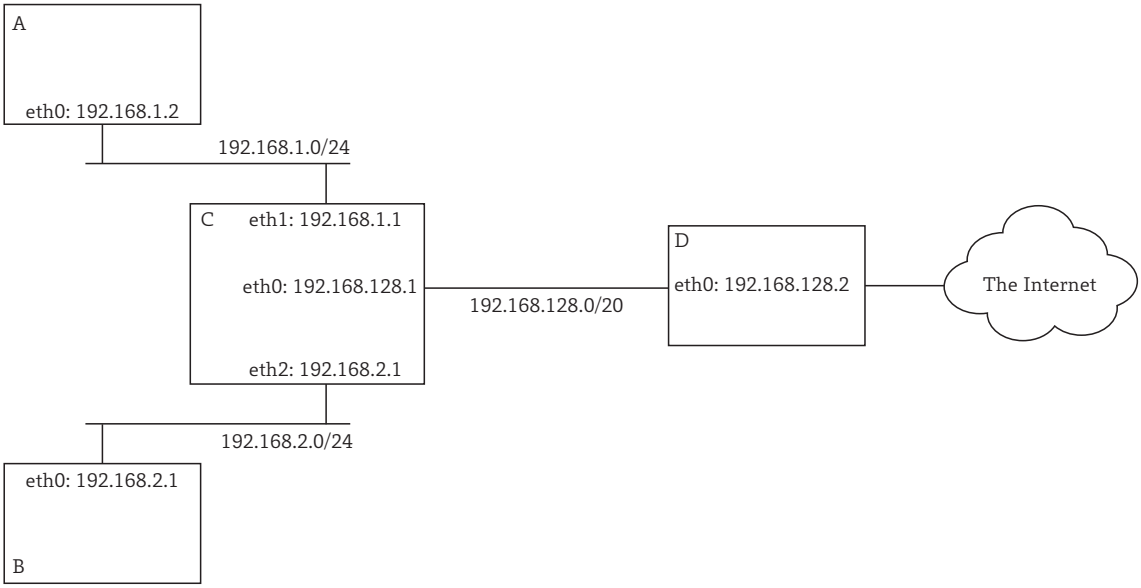

図 6　IPv4 ネットワーク例

リプライへのポインタ arph である。10〜14 行目は受け取った ARP リプライを標準出力へ出力している。17 行目は受け取った ARP リプライの情報を ARP テーブルに追加している。21 行目以降が送信バッファ中にある IPv4 パケットを送信する処理となる。21 行目で送信バッファを走査して IPv4 パケットを取り出し、24 行目で取り出した IPv4 パケットの宛先 IPv4 アドレスが解決されているかを検索し、検索が失敗した場合は次のパケットへ処理を移す（25〜27 行目）。31〜34 行目で、宛先 MAC アドレスを設定し Ethernet フレームを送信している。37 行目以降は、送信済みのバッファを解放する処理となる。

重要ポイント

- ARP を利用して IPv4 アドレスと MAC アドレスの解決が行われる
- OS 内部には IPv4 アドレスと MAC アドレスの対応を記録した ARP テーブルが存在する
- ARP テーブルは ARP パケットを受信したタイミングで更新される

1.7　IPv4

本節ではネットワーク層のプロトコルとして IPv4 を中心に議論を行う。IPv4 以外のネットワーク層のプロトコルとしては、IPv6 があるが、本稿での説明は割愛する。ネットワーク層のプロトコルは IPv4 と IPv6 以外にも、IPX などのプロトコルがあったが現在ではほとんど使われていない。

IPv4 の役目を一言で言うならば、世界中のコンピュータを一意に識別して相互接続性を実現することである。IPv4 の識別子としては 32 ビットの IPv4 アドレスが用いられ、相互接続性はルーティングと呼ばれるパケット転送のメカニズムを用いて実現される。したがって、全ての IPv4 ノードは、IPv4 アドレスと IPv4 のルーティングテーブルを持つことになる。

図 6 は、ある IPv4 ネットワークのトポロジ図と各ノードの IPv4 アドレス割当を示している。この図では 4 つのノードが存在し、中央のノード C と、D がルーティングを行っている。また、ノード D がインターネットに接続されており、ノード A、B、C はノード D を経由してインターネットへ接続する。

表 3　IPv4 アドレスの表記

CIDR 表記	ネットワークアドレス	ブロードキャストアドレス	サブネットマスク
192.168.1.0/24	192.168.1.0	192.168.1.255	255.255.255.0
172.16.0.0/16	172.16.0.0	172.16.255.255	255.255.0.0
10.0.0.0/8	10.0.0.0	10.255.255.255	255.0.0.0

表 4　ノード C のルーティングテーブル

宛先ネットワーク	次ホップ IPv4 アドレス	インターフェース
0.0.0.0/0	192.168.128.2	eth0
192.168.128.0/20		eth0
192.168.1.0/24		eth1
192.168.2.0/24		eth2

ノード中にある「eth0: 192.168.1.2」という表記は、このノードには eth0 と言う名で識別される物理的なネットワークのポートがあり、eth0 のポートには 192.168.1.2 と言う IPv4 アドレスが割り当てられていると言う事を意味している。ノード間を接続する線は IPv4 ネットワークのネットワークを表しており、このネットワークのアドレスは 192.168.1.0/24 というように表記される。

インターネットでは、IP アドレスを複数のネットワークに分割して分散的に管理が行われる。この分割されたネットワークのことをサブネットと呼ぶ。サブネットのアドレスは、先に示したように 192.168.1.0/24 と表記されるが、これは、上位 24 ビットまでがサブネットのネットワークアドレスであると示している。つまり、この場合、192.168.1.0 がネットワークアドレスになる。192.168.1.0/24 という表記は Classless Inter-Domain Routing (CIDR、読み方はサイダー) 表記とも呼ばれる。192.168.1.0/24 という CIDR 表記は、192.168.1.0 というネットワークアドレス表記と、255.255.255.0 というサブネットマスクと呼ばれる表記と等価である。これはつまり、/24 は、上位 24 ビットに 1 が立っているビット列と解釈されるためである。

CIDR の C は Classless という意味であったが、これはつまりクラスのあるサブネットの構成方法もあるということである。クラスのある場合、サブネットマスクがそれぞれ、255.0.0.0、255.255.0.0、255.255.255.0 であるサブネットを、クラス A、クラス B、クラス C のネットワークと識別する。以前はクラスという考え方でサブネットを構成していたが、IPv4 のアドレス空間を効率的に利用するため、現在はクラス無しの CIDR 方式でサブネットを構成する。

IPv4 にはブロードキャストアドレスと呼ばれるアドレスがある。192.168.1.0/24 いうアドレスがあった場合は、ネットワークアドレスに下位 $32 - 24 = 8$ ビットに 1 がたったアドレスがブロードキャストアドレスになる。つまりこの場合、192.168.1.255 がブロードキャストとなる。また、255.255.255.255 もブロードキャストアドレスであるが、これはローカルネットワークのブロードキャストアドレスになる。図 6 のノード A を例に取ると、ノード A が 255.255.255.255 を宛先に IPv4 パケットを eth0 に送信した場合、受信はノード A の eth0 と、ノード C の eth1 によって行われる。

表 3 は IPv4 アドレス表記の例を示している。例えば、172.16.0.0/16 と表記された場合は、172.16.0.0 がネットワークアドレスとなり、ネットワークの下位 16 ビットすべてに 1 がたった 172.16.255.255 がブロードキャストアドレスとなり、/16 のサブネットマスクは 255.255.255.0 となる。

次に、IPv4 のルーティングとルーティングテーブルについて説明する。ルーティングとはすなわちパケッ

表 5　ノード C のルーティングテーブル

アドレス	2 進数表記	マッチビット数
192.168.1.2	11000000 10101000 00000001 00000010	
192.168.128.0/20	11000000 10101000 1000	16
192.168.1.0/24	11000000 10101000 00000001	24
192.168.2.0/24	11000000 10101000 00000002	23

ト転送のことであり、ルーティングテーブルはルーティングを行うためのルールを記したテーブルのことである。表 4 は図 6 のノード C のルーティングテーブルである。IPv4（と IPv6）では、最長プレフィックスマッチ（longest prefix match）方式と呼ばれる方式でパケットの転送を行う。例えば、192.168.1.2 宛のパケットをノード C が eth1 のインターフェースから受信した場合、192.168.128.0/20 の先頭 20 ビット、および、192.168.1.0/24 と 192.168.2.0/24 の先頭 24 ビットを、192.168.1.2 の先頭ビットからマッチさせたとき最も一致する長さが長いエントリが選ばれる。

この最長プレフィックスマッチを表したのが表 5 となる。この表では、IPv4 アドレス及びそれを 2 進数表記した数値が示されており、192.168.1.2 と最上位ビットからマッチさせた結果、マッチしたビット数が 3 列目に示される。この表で示されるように、192.168.128.0/20 と 192.168.1.2 を最長プレフィックスマッチさせると上位 16 ビットがマッチし、同じように 192.168.1.0/24 と 192.168.2.0/24 に対して 192.168.1.2 を最長プレフィックスマッチさせると、192.168.1.0/24 は上位 24 ビット、192.168.2.0/24 は 23 ビットマッチする。したがって、表 4 のルーティングテーブルを用いると、192.168.1.2 宛のパケットは eth1 へ転送される。

表 4 で宛先ネットワークが 0.0.0.0/0 と示されるエントリは、デフォルト経路と呼ばれるものとなる。デフォルト経路とは、デフォルト経路以外のエントリとマッチしなかった時に転送される経路である。例えば、172.16.0.1 宛のパケットの転送先を表 4 で検索した場合、0.0.0.0/0 以外には 1 ビットもプレフィックスマッチしないため、デフォルト経路が選択される。

では、もう一度、図 2 を見返してみよう。この図では、データリンク層のアドレスである MAC アドレスが書き換えられてパケットが転送される様子が示されている。図 2 の L3 スイッチが、図 6 のノード C に相当するが、この図で示されるように、ルーティングテーブルで次ホップアドレスを検索した後、宛先 MAC アドレスが次ホップの MAC アドレスに設定されてパケット（正確に言うとパケットを含むフレーム）が転送される。ちなみに、IP レベルでのデータ転送を行う機器を L3 スイッチまたはルータと呼ぶ。

続いて、IPv4 ヘッダを表す ip 構造体の説明を行う。ip 構造体はソースコード 13 のように定義されている。

ソースコード 13　IPv4 ヘッダ定義 (/usr/include/netinet/ip.h)

```
1  /*
2   * Structure of an internet header, naked of options.
3   */
4  struct ip {
5  #if _BYTE_ORDER == _LITTLE_ENDIAN
6          u_int     ip_hl:4,              /* header length */
7                    ip_v:4;               /* version */
8  #endif
9  #if _BYTE_ORDER == _BIG_ENDIAN
10         u_int     ip_v:4,               /* version */
11                   ip_hl:4;              /* header length */
12 #endif
13         u_int8_t  ip_tos;               /* type of service */
```

```
14          u_int16_t ip_len;               /* total length */
15          u_int16_t ip_id;                /* identification */
16          u_int16_t ip_off;               /* fragment offset field */
17  #define IP_RF 0x8000                    /* reserved fragment flag */
18  #define IP_DF 0x4000                    /* dont fragment flag */
19  #define IP_MF 0x2000                    /* more fragments flag */
20  #define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
21          u_int8_t  ip_ttl;               /* time to live */
22          u_int8_t  ip_p;                 /* protocol */
23          u_int16_t ip_sum;               /* checksum */
24          struct    in_addr ip_src, ip_dst; /* source and dest address */
25  };
```

7 または 10 行目にある ip_v メンバ変数は、IP のバージョンを表すための変数であり、IPv4 の場合は先頭 4 ビットの値は必ず 4 となる。C 言語のビットフィールドを利用する際にはエンディアンを考慮する必要があるが、ここでは 4〜12 行目のようにマクロでエンディアン指定を行っている。6 または 11 行目の ip_hl メンバ変数は IP ヘッダ長を表すための変数であり、この値は 4 オクテット単位（32 ビット単位）の値となる。例えば、ヘッダ長が 20 バイトの場合、20/4=5 が ip_hl 変数に格納される。

13 行目の ip_tos メンバ変数は Quality of Service (QoS) や DiffServ [5] などで利用される変数である。TOS は Type of Service の略であり、パケット転送時の優先順位などを行うために利用されるが、詳細は本書の範囲を超えるため割愛する。

14 行目の ip_len メンバ変数は IP パケットの合計長を表すための変数である。15、16 行目の ip_id と ip_off メンバ変数はパケットがフラグメントされた時に利用される変数である。ip_id は複数のフラグメント化されたパケットを識別するために利用される。例えば、1 つのパケットが 2 つのパケットに分割された場合、ip_id に同一の値が格納される。ip_off は上位 3 ビットがフラグメントフラグを指定するために利用され、下位 13 ビットがオフセットを指定するために利用される。フラグメントフラグは、17〜19 行目で示されるように、ip_off と IP_DF（0x4000）の論理積が 0 でない場合はフラグメント禁止で、ip_off と IP_DF（0x2000）の論理積が 0 でない場合はフラグメントパケットが続くことを意味する。ip_off の下位 13 ビットは、そのパケットはフラグメントする前のパケットから何バイト目かを示す値となる。

21 行目の ip_ttl メンバ変数（ttl は Time to Live の略）は IP パケットの生存時間を示すために用いられる。先に説明したように、IP は複数のルータを経由して転送されるが、このとき経路にループがあった場合に永遠にパケットが転送され続けてしまう。このような事態を防ぐために、ルータはパケットを転送する際に ip_ttl の値を 1 減算してから転送し、ip_ttl の値が 1 の場合は転送しないようにする。

22 行目の ip_p メンバ変数は IP パケットの上にあるトランスポートプロトコルを示すために用いられる。例えばこの値が IPPROTO_TCP（6）の場合は TCP プロトコル、IPPROTO_UDP（17）の場合は UDP プロトコルで有ることを示す。ちなみに、IPPROTO_TCP や IPPROTO_UDP は netinet/in.h ヘッダで定義されている。

23 行目の ip_sum メンバ変数は IP ヘッダのチェックサムを格納するために用いられる。なお、IP パケット全体ではなく、IP ヘッダのみであることに注意されたい。

つづいて、ソースコード 14 で示される、IPv4 パケットの入力処理を行う ipv4_input 関数について解説する。ipv4_input 関数は、IPv4 パケットを受け取り、転送、あるいは IP より上位の層である TCP または UDP の関数へ処理を渡す関数となっている。ipv4_input 関数の取る引数 iph は、IPv4 パケットの先頭アドレスを指す ip 構造体へのポインタである。

ソースコード 14　ipv4_input 関数

```
1  /*
2   * IPv4 パケット入力関数
3   * 引数:
4   *   iph: 入力パケット
5   */
6  void ipv4_input(struct ip *iph) {
7      // 自ホスト宛かチェック
8      if (!is_to_me(iph)) {
9          if (IS_L3BRIDGE) { // L3 ブリッジが有効かチェック
10             // TTL が 1 以下なら転送しない
11             if (iph->ip_ttl <= 1)
12                 return;
13
14             // 自ホスト宛でない場合転送
15             ipv4_forward(iph);
16         }
17         return;
18     }
19
20     uint8_t *nxt = (uint8_t *)iph;
21     nxt += iph->ip_hl * 4;
22
23     switch (iph->ip_p) {
24     case IPPROTO_TCP:
25         tcp_input((struct tcphdr *)nxt);
26         break;
27     case IPPROTO_UDP:
28         udp_input((struct udphdr *)nxt);
29         break;
30     default:;
31     }
32 }
```

ソースコード 14 の 8〜18 行目では、IPv4 パケットの転送に関する処理が行われる。ここでは、まず 8 行目で自ホスト宛のパケットでないことをチェックし、9 行目で L3 ブリッジ、つまりルーティング機能が有効化をチェックしている。パケットが自ホスト宛ではなく、ルーティング機能が有効であれば、IPv4 パケットの転送処理を行う。11 行目では、転送の前に IPv4 パケットの TTL をチェックし、TTL が 1 以下の場合は 12 行目で転送処理を終了させる。TTL をチェックした後、15 行目で実際に転送を行い、転送を行った後に 17 行目で処理を終了させる。

20 と 21 行目では、次のプロトコルへのポインタ、つまり、IP パケットのデータ部分へのポインタを取得している。ip 構造体中の ip_hl メンバ変数は 4 バイト単位の値となっているため、21 行目では ip_hl から 4 を乗算した先を取得している。

23 から 31 行目では、IP より上位層へ処理を渡している。上位の層にどのようなプロトコルのデータが入っているかは、ip 構造体中の ip_p メンバ変数に格納されているため、23 行目ではこの値に基づき処理を分岐させる。ip_p の値が IPPROTO_TCP の場合は、24〜25 行目に処理がうつり、TCP の処理を行う関数へわたされる。また、ip_p の値が IPPROTO_TCP の場合は、27〜28 行目に処理がうつり、UDP の処理を行う。ここでは TCP と UDP の処理しか書かれていないが、実際のネットワークスタックでは ICMP などの処理も行われる。

つづいて、ソースコード 15 で示される、IPv4 パケットの転送処理を行う ipv4_forward 関数について解説

を行う。ipv4_forward 関数の取る引数 iph は、ipv4_input 関数と同じく IP パケットへのデータを指す ip 構造体へのポインタである。

ソースコード 15　ipv4_forward 関数

```
/*
 * ルーティングテーブルに基づいてIPパケットの転送を行う関数
 * 引数:
 *   iph: 転送するIPパケットへのポインタ
 */
static void ipv4_forward(struct ip *iph) {
    // ルーティングテーブルから宛先インターフェースを決定
    struct rtentry *entry = route_lookup(&iph->ip_dst);
    if (entry == NULL) // 宛先がルーティングテーブルに存在しない
        return;

    // 自分宛てのパケットかを検査
    if (entry->ifp != NULL && entry->ifp->addr.s_addr == iph->ip_dst.s_addr) {
        ipv4_input(iph);
        return;
    }

    uint32_t len = ntohs(iph->ip_len);      // IPパケット長
    uint32_t ethlen = ETHER_HDR_LEN + len; // Ethernetフレーム長

    iph->ip_ttl--; // TTLを1減算

    // チェックサムを更新
    iph->ip_sum = 0;
    iph->ip_sum = cksum(iph, iph->ip_hl * 4);

    uint8_t buf[ethlen]; // 一時バッファ
    struct ether_header *eh = (struct ether_header *)buf;

    // Ethernetヘッダおよび、IPヘッダへアドレスを設定
    memcpy(eh->ether_shost, entry->ifp->ifaddr,
           ETHER_ADDR_LEN);                 // 送信元MACアドレス
    memcpy(buf + ETHER_HDR_LEN, iph, len); // IPヘッダ
    eh->ether_type = htons(ETHERTYPE_IP);  // Ethernetタイプを IPに設定

    // 次ホップIPアドレスを決定
    struct in_addr nextip =
        (entry->addr.s_addr == 0) ? iph->ip_dst : entry->addr;

    // 宛先IPアドレスの宛先 MACアドレスを ARPテーブルから検索
    struct ip2mac *mac = find_mac(&nextip);
    if (mac == NULL) {
        // ARPテーブルにないため、宛先 IPアドレスに対応する MACアドレスを問い合わせ
        arp_req(entry->ifp, &nextip);

        // 送信バッファにコピー
        add2sendbuf(eh, ethlen, &nextip);
    } else {
        // 宛先MACアドレスを設定
        memcpy(eh->ether_dhost, mac->macaddr, ETHER_ADDR_LEN);

        // インターフェースへ出力
        ether_output(entry->ifp, eh, ethlen);
```

```
54      }
55  }
```

ソースコード 15 の 8 行目では、ルーティングテーブルからパケットの宛先アドレスをキーとして転送先を検索している。もし、転送先がルーティングテーブルから見つからない場合は、9〜10 行目で宛先不明のパケットとして処理を終了させる。ルーティングテーブルの検索は、route_lookup 関数をもちいて行うが、ここではその詳細は省略する。

13〜16 行目では、パケットの宛先が自身宛かを検査し、自信宛の場合は ipv4_input 処理へ以降する。自信宛のパケットの検査は、インターフェースに付与されている IPv4 アドレスとパケットの宛先が同じであるかどうか検査することで可能である。

18 行目では IP パケット長を ip 構造体の ip_len メンバ変数から取得し、そこから 19 行目で Ethernet フレーム長の計算を行う。また、21 行目では TTL の減算を行い、24、25 行目では IP ヘッダのチェックサムを計算している。

27〜34 行目は、Ethernet フレームの構築を行っている。[*3]27 行目でバッファを確保し、28 行目で Ethernet フレームへのポインタに変換している。31 行目では Ethernet フレームの送信元 MAC アドレスを設定し、32 行目で Eternet フレームへ IP パケットの情報をコピーし、33 行目で Ethernet フレームタイプを IPv4 パケットであると設定している。

37、38 行目では、次ホップの IP アドレスを決定してる。次ホップの IP アドレスは、自身の持っているインターフェースが、パケット宛先ネットワークに所属している場合は、パケットの宛先が次ホップ IP アドレスとなり、そうでない場合はルーティングテーブルのルックアップ結果から得られたアドレスが次ホップの IP アドレスとなる。この実装では、ルーティングテーブルが示す次ホップが 0 の場合は、インターフェースが所属するネットワークの宛先であることを示すような実装にしているため[*4]、37 行目でルーティングテーブルの示す次ホップアドレスの値を検査している。

41〜54 行目は Ethernet フレームの転送を行う関数である。ここでは、41 行目で宛先 MAC アドレスを次ホップ IP アドレスから検索しているが、検索結果がない場合は、42〜47 行目の処理に、ある場合は 49〜53 行目の処理にうつる。次ホップ IP アドレスの MAC アドレスが得られない場合は、ARP を用いて MAC アドレスの解決を行う必要があり、44 行目で ARP リクエストを行う関数を呼び出している。実際の Ethernet フレーム送信は、ARP の結果が返ってきたあとに行われるため、47 行目で一旦バッファに Ethernet フレームを退避している。次ホップ IP アドレスの MAC アドレスが得られた場合は、50 行目で Ethernet フレームの宛先 MAC アドレスを設定し、53 行目で ether_output 関数を呼び出し Ethernet フレームの送信を行う。

ソースコード 16 は、IPv4 パケットの送信を行うための ipv4_output 関数である。この関数は単純に、送信したい IP パケットへのデータを指す ip 構造体へのポインタ iph を受け取り、TTL を設定した後に ipv4_forward 関数を呼び出している。

ソースコード 16　ipv4_output 関数

```
1  /*
2   * IPv4 出力関数
```

[*3] よく考えると、IP スタックと Ethernet スタックの独立性を保つためには、ここで Ethernet フレームの構築を行うのは不適切であり、ether_output 関数で Ethernet フレームの構築を行うべきである。しかし、そうするとルーティングテーブルルックアップの結果を ARP 応答まで保持しておかなければならず、同期処理、ガベージコレクション処理の実装が複雑になるため、実装を簡単にするためにこのようにした。

[*4] インターフェースの作成時に次ホップが 0 の経路がルーティングテーブルに自動的に追加されるようにしている。

```
 3  * 引数:
 4  *   iph: 出力パケット
 5  */
 6 void ipv4_output(struct ip *iph) {
 7     iph->ip_ttl = 32;
 8     ipv4_forward(iph);
 9 }
```

重要ポイント

- IPv4 ではサブネットと呼ばれる単位でネットワークを分割して管理する
- サブネットワークのアドレスは IPv4 アドレス（例：192.168.0.0）とサブネットマスク（255.255.255.0）で表現されるが、これは CIDR 表記で表現されることもある（例：192.168.0.0/24）
- ルーティングテーブルは、宛先ネットワークアドレス、サブネットマスク、次ホップアドレス、インターフェースの情報で構成される
- ルーティングテーブルの検索は、最長プレフィックスマッチで行われる

2 演習問題

本節では、仮想マシンを用いてネットワークとファイアウォールの設定を行う問題を提示する。演習環境は Vagrant+VirtualBox を用いて構築することが出来る。演習用の OS は OpenBSD としている。OpenBSD を選択した理由は、OpenBSD はセキュリティが強く意識された OS であり、OpenBSD の pf の設定は大変明瞭でわかりやすく、初学者に向いているためである。pf の設定がわかれば、実際の商用ネットワーク機器の設定も行うことが出来るようになるだろう。

演習環境は、mkenv/vagrant 以下にある各ディレクトリ内で vagrant を用いると構築することが出来る。例えば、以下のようにすることで問題 1 の演習環境を構築することが出来る。

```
$ cd mkenv/vagrant/quiz01
$ vagrant up
```

問題 1 では、3 つのノードがあるが、そのうちのノード h1 へログインしたいときは下記のように入力するとログインできる。

```
$ vagrant ssh h1
```

他のノードへログインしたい場合は、一旦 h1 から抜けて、同じように h2 や h3 へとログインすれば良い。演習環境を一旦終了させたいときは、halt コマンドを実行すると終了させることができる。

```
$ vagrant halt
```

仮想マシンが起動したままだと、メモリや CPU リソースを消費するので、一旦他の作業をしたい場合は、仮想マシンを停止しておくとよいだろう。もしも問題をとき終わって環境を破棄したくなったり、設定をミスってしまってもう一度演習環境を初期状態に戻したい場合は、destroy で破棄することが出来る。

```
$ vagrant destroy
```

いらなくなった環境は、そのまま放って置くとディスクスペースを圧迫するため、必要なくなったら削除しておこう。

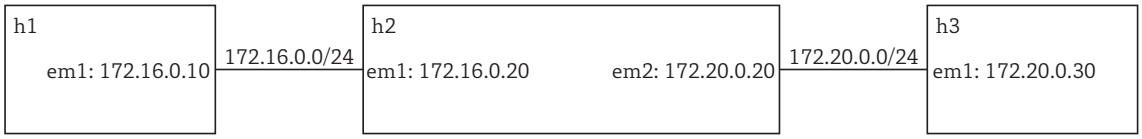

図 7　問題 1：ルーティング実験の 3 ノードトポロジ図

ソースコード 17 は、問題 1 の環境を構築するための Vagrantfile である。Vagrant は Ruby によって実装されおり、Vagrantfile というスクリプトで環境設定を記述する。この設定ファイルでは、仮想マシンに OpenBSD version 6 を利用した、h1、h2、h3 というホストを用意することが記述されている。また、L2 ネットワークは、q01_a、q01_b という 2 つのネットワークを利用するようにしている。

ソースコード 17　演習問題 1 の Vagrantfile

```
1  Vagrant.configure("2") do |config|
2    config.vm.box = "generic/openbsd6"
3
4    config.vm.define :h1 do | h1 |
5      h1.vm.hostname = "h1"
6      h1.vm.network "private_network", ip: "172.16.0.10", virtualbox__intnet: "q01_a"
7    end
8
9    config.vm.define :h2 do | h2 |
10     h2.vm.hostname = "h2"
11     h2.vm.network "private_network", ip: "172.16.0.20", virtualbox__intnet: "q01_a"
12     h2.vm.network "private_network", ip: "172.20.0.20", virtualbox__intnet: "q01_b"
13   end
14
15   config.vm.define :h3 do | h3 |
16     h3.vm.hostname = "h3"
17     h3.vm.network "private_network", ip: "172.20.0.30", virtualbox__intnet: "q01_b"
18   end
19 end
```

2.1　問題 1：3 ノードの単純なネットワーク

図 7 のようなネットワークがあるとする。このとき、ノード h1 からノード h3 へ及び、ノード h3 からノード h1 へ IPv4 パケットが到達するように、ノード h1〜h3 を設定せよ。ただし、ブリッジ接続ではなく、ルーティングテーブルを設定して L3 レベルでの設定を行うこと。ヒント：ノード h1〜h3 のルーティングテーブルを route コマンドで設定し、ノード h2 を IPv4 のフォワーディングを行うように sysctl コマンドで設定せよ。

2.2　問題 2：4 ノードのネットワーク

図 8 のようなネットワークがあるとする。このとき、各ノードから全てのノードへ IPv4 パケットが到達するように、ノード h1〜h4 を設定せよ。ただし、ブリッジ接続ではなく、ルーティングテーブルを設定して L3 レベルでの設定を行うこと。

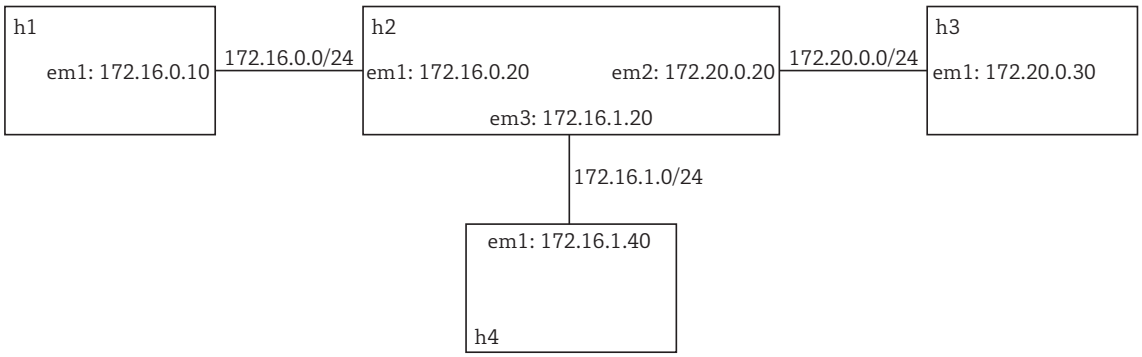

図 8　問題 2 および 3：ルーティング実験、DMZ 実験の 4 ノードトポロジ図

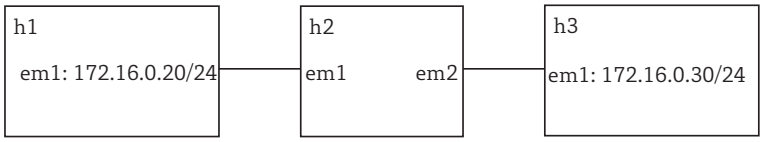

図 9　問題 4：ブリッジ実験のトポロジ図

2.3　問題 3：DMZ（DeMilitarized Zone）

同じく、図 8 のネットワークを利用して、以下の要求を満たすようにサービスを構築し、ノード h2 に pf でファイアウォールを構築せよ。ファイアウォールの設定後は、ping、curl コマンドなどを利用して設定が要求どおりとなっているか確認すること。

- ノード h1 の所属するネットワーク 172.16.0.0/24 を社内ネットワークとする
- ノード h3 の所属するネットワーク 172.16.20.0/24 をインターネットに接続するための対外ネットワークとする
- ノード h4 の所属するネットワーク 172.16.1.0/24 を対外向けサービスを設置するための DMZ ネットワークとする
- ノード h1、h3、h4 に Web サーバを構築せよ
- 社内ネットワークから対外ネットワーク及び DMZ ネットワークへの TCP、UDP、ICMP 接続が可能
- DMZ ネットワークから社内ネットワークへの TCP、UDP、ICMP 接続は禁止
- DMZ ネットワークから対外ネットワークへの TCP、UDP、ICMP 接続は可能
- 対外ネットワークから DMZ ネットワークへの接続は Web のみ可能
- 対外ネットワークから社内ネットワークへの TCP、UDP、ICMP 接続は禁止
- 社内ネットワークと DMZ から対外ネットワークへむけて、ソース IP アドレスを詐称してパケットを送信することは禁止

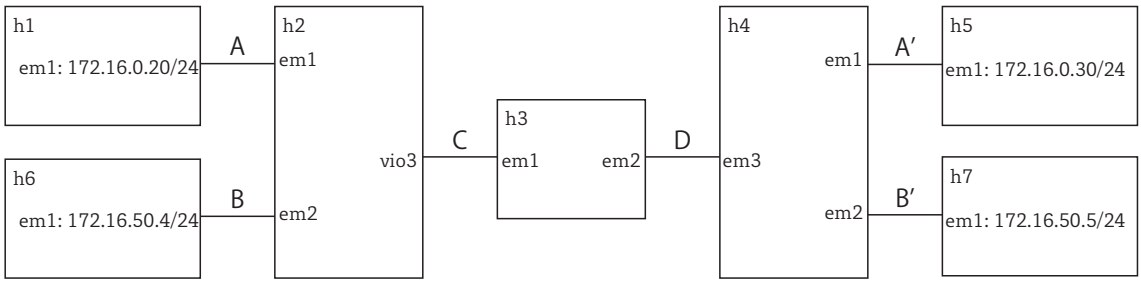

図 10　問題 5：VLAN 実験のトポロジ図

2.4　問題 4：ブリッジ

図 9 のネットワークがあるとする。このとき、ノード h2 でブリッジ設定、ノード h1 と h3 を互いに通信できるようにせよ。ヒント：ブリッジの設定は ifconfig コマンドで行える。

2.5　問題 5：VLAN

図 10 のネットワークがあるとする。このとき、リンク A と A' が同じセグメントに、また、リンク B と B' が同じセグメントとなるように、タグ VLAN とブリッジの設定をノード h2、h3、h4 で行え。ヒント：タグ VLAN とブリッジの設定は ifconfig コマンドで行える。

2.6　問題 6：NAPT

問題 3 のネットワーク設定を改変し、社内ネットワークから対外ネットワークへの接続は NAPT で接続するように修正せよ。

参考文献

[1] J. Postel. Internet protocol, September 1981. RFC0791.

[2] Internet Assigned Number Authority (IANA). Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[3] Internet Assigned Number Authority (IANA). IEEE 802 Numbers. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[4] J. Postel. Echo protocol, May 1983. RFC0862.

[5] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers, December 1998. RFC2474.