QualNet
Building Smarter Networks

QualNet Hacks

＃４ルーティング

QualNet
Building Smarter Networks

About QualNet Hacks

QualNet は、ほぼ全てソースコードが公開されています。

これらのソースコードには QualNet の内部を理解するために有益な情報が沢山ちりばめられています。

しかしながら、ソースコードの量は莫大であり、その内容を簡単に理解することが難しいのも事実です。

本書は、QualNet 内部の理解をより深めて頂くことを目的として作成しました。

本書を手掛かりにして、より一層 QualNet 活用して頂ければ幸いです。

このドキュメントは QualNet5.1 のソースコードに準拠します

【ソースコードに関する注意事項】
本ドキュメントには、ソースコードの一部が複写されています。ソースコードの使用に関しては、以下の開発元の制限に則りますので、ご注意下さい。

contents

4 ルーティング

4.1 ルーティング概要

一言に「ルーティング」と言ってもその定義は実は難しく、現実の世界では様々な文脈で「ルーティング」という用語が用いられている。例えば、IP パケットの送信時に経路表を引く行為をルーティングと呼ぶ場合もあれば、経路表を引いて実際に次ホップに向けて送信する行為までを含める場合もある。また、経路表を構築・管理する行為をルーティングと呼ぶ場合もあれば、それも含めて経路表に関係する一切の処理をまとめてルーティングと呼ぶ場合もある。「ルーティング」の中心的存在である経路表についてもその呼び名は様々であり、例えばルーティングテーブルと呼ばれたり、フォワーディングテーブルと呼ばれたり FIB（Forwarding Information Base）と呼ばれるときもある。

本書では、これらの処理やデータ構造を明確に区別して説明するために、可能な限り「ルーティング」という用語は用いない解説を試みる。以後の解説では、図 4-1 に示した用語を用いる。

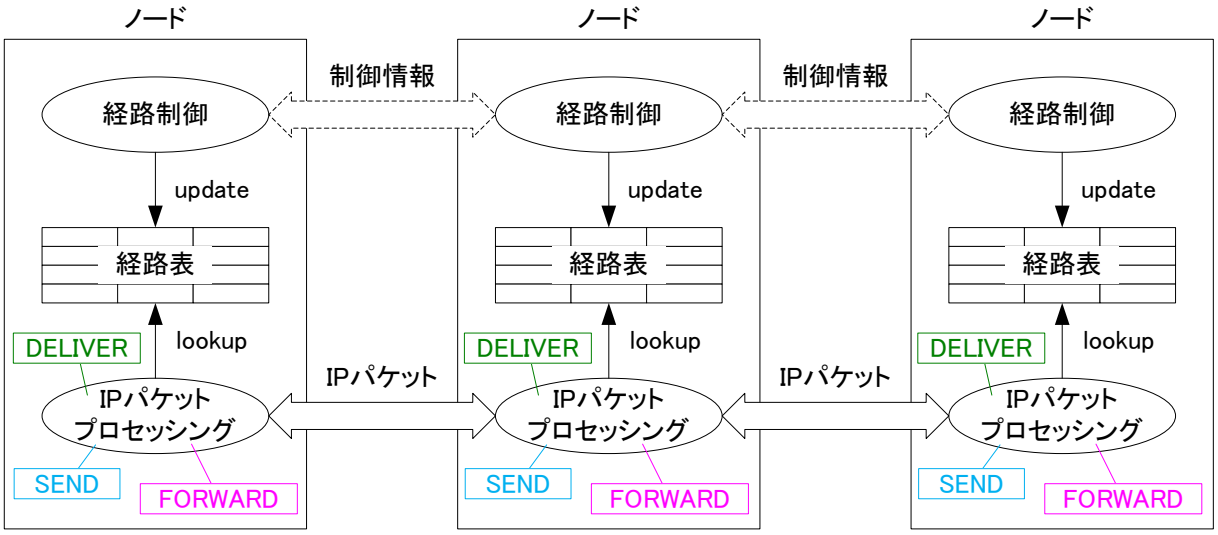

図 4-1 本書におけるルーティング関連用語の整理

まず経路情報を格納したテーブルを本書では「経路表」と呼ぶことにする。それから、この経路表の構築・管理を行う作業を本書では「経路制御」と呼ぶことにする。さらに、自身が受信したデータパケットならびに自身から発生するデータパケットに対して IP レイヤで行われる送信・受信・転送に関わる処理をまとめて本書では「IP パケットプロセッシング」と呼ぶことにする。

経路表は宛先アドレスに対して outgoing インタフェースと nexthop アドレスを紐付けた情報の集合であり、ノードに1つ存在する場合が一般的である。（場合によっては経路制御プロトコルごとに固有の経路表を独立して持つ場合もあったり、サービスクラス毎に異なる経路表が用意される場合もある。）

経路制御は経路表エントリの追加・更新・削除を行うこと（経路表の update）が主要な役目であり、コンフィグレーションあるいはオペレータによる人力操作によって与えられた経路エントリを反映する静的なものから、ノード間で制御情報を交換して動的に経路エントリの追加・削除を行うものまで存在する。動的な制御情報の交換を行う際の手順やルールを経路制御プロトコルと呼ぶ。この経路制御プロトコルにはプロアクティブ型のものとリアクティブ型のものが存在する。プロアクティブ型のものはノード間で IP パケットプロセッシングを介してやりとりされるデータパケットとは独立に制御情報を交換しながら経路表を更新するのに対して、リアクティブ型のものは IP パケットプロセッシングと連動し経路探索の必要が生じた時点でオンデマンドにノード間で制御情報を交換して経路表を更新するという違いがある。

IP パケットプロセッシングは、上位レイヤから渡された自ノード発パケットの処理（SEND 処理）と下位レ

イヤから渡された他ノードからの受信パケットの処理(RECEIVE 処理)で構成される。RECEIVE 処理は、受信パケットが自ノード宛か他ノード宛かによってさらに上位レイヤへ引き渡す処理(DELIVER 処理)と他ノードに転送する処理(FORWARD 処理)に分岐する。そして、SEND 処理および FORWARD 処理において経路探索(経路表の lookup)が行われる。

これ以降、QualNet でのルーティング処理の実装を「IP パケットプロセッシング」、「経路表」、「経路制御」それぞれについて説明してゆくが、説明の都合上、その前にシミュレーション特有の要素であるルータ機器特性のモデル化方法について説明を行う。

4.2 ルータモデル

異なるネットワークをつなげる機器、つまりネットワークからネットワークへ IP パケットの転送(=FORWARD 処理)を行う機器を「ルータ」と呼ぶ。一般的に OS(Unix,Linux,Windows,...)カーネル自体がパケットの転送機能を持つため、これらの OS がインストールされた通常の PC も広義のルータと捉えることができる。ただし、OS カーネル単体では経路制御の機能は持っておらず、routed, Quagga 等の経路制御デーモンアプリケーションと組み合わせることでソフトウェア的に経路制御機能を付加することができる。一方でルータ専用の機器も多数存在し、一般的にルータと言えばこちらのことを指すことが多いのではないだろうか。専用機器には、前述の一連のルーティング関連処理を行うための専用チップが搭載され、ソフトウェア的な処理に比べ大容量・帯域のパフォーマンスを実現している。

IP パケットプロセッシングのパフォーマンス(スループット,遅延 etc)は、その機器特性(処理遅延やアーキテクチャ種別)に大きく依存する。さらに IP パケットプロセッシングのパフォーマンスはネットワーク全体の挙動に大きく影響を与えてくる。したがって、シミュレータ上で現実的なネットワーク(ボトルネック等)を模擬し種々の検証を行う上で、ルータの処理性能のモデル化は重要な要素になってくる。

QualNet では、ルーティング処理のロジックに加えてこれらルータの機器特性を模擬する手段を「ルータモデル」という名称で提供している。

4.2.1 ルータ機器性能のモデル化

現実の世界において、パケットがルータやスイッチなどの通信機器を通過する際には、その機器性能に応じた処理遅延が発生する。一方で QualNet シミュレーションのような離散時間シミュレーションの世界では、普通にパケット処理関数を呼び出す限りにおいては、どれだけ重たい処理をその関数内で行ってもシミュレーションの世界の中でのパケット処理遅延が大きくなることはなく、単にシミュレーションが重くなるという結果にしかならない。QualNet では、機器性能を考慮に入れたシミュレーションを実現するために、キューを用いた遅延発生メカニズムを IP レイヤ内に(プロトコルスタックの背後に潜ませるかたちで)設けることでその模擬を実現している。このメカニズムは一般的なスイッチのバックプレーンの処理をモデル化したものであり、これを QualNet ではルータモデルと呼んでいる。表 4-1 にルータモデルに関連する設定パラメータの一覧を示す。

表 4-1 ルータモデル設定パラメーター一覧

パラメータ	概要
ROUTER-MODEL	モデル名
ROUTER-BACKPLANE-TYPE	ルータバックプレーンの処理タイプ。 CENTRAL/DISTRIBUTED のどちらかを選択。
ROUTER-BACKPLANE-THROUGHPUT	ルータバックプレーンの処理速度。単位は[bps]。
ROUTER-PERFORMANCE-VARIATION	パフォーマンスの変動量。%で与える。

また、ROUTER-MODEL-CONFIG-FILE というパラメータで、ルータモデル定義ファイルを与えておけば、上記パラメータのうち ROUTER-MODEL のみを指定することで、続くパラメータを設定させることが出来る。なお、下記ディレクトリに現実の各種ルータ製品の公称値が反映されたルータモデル定義ファイルが用意されており、現実のルータ製品に沿ったモデルを選んで設定することが可能である。

```
$QUALNET_HOME/scenarios/default/default.router-models
```

4.2.1.1 ルータバックプレーン処理

設定で与えるバックプレーン容量(ROUTER-BACKPLANE-THROUGHPUT)と処理対象パケットのサイズから、当該パケットにかかる処理遅延時間が算出できる。QualNet のルータモデルでは、処理パケットを一旦バックプレーンに用意したキュー(以下、バックプレーンキューと呼ぶ。"CPU Queue"と"Input Queue"の 2 種類がある。)に格納しておき、算出された遅延時間後に当該キューに対するデキューイベントが発生するようにすることで、処理遅延の模擬を実現している。

バックプレーンの種類としては、Central と Distributed の 2 タイプが設定可能である。それぞれバス型のバックプレーンとクロスバースイッチ型のバックプレーンをモデル化したものと考えればよい。Central タイプの場合は SEND 処理対象のパケットおよびすべてのネットワークインタフェースにおける FORWARD／DELIVER 処理対象のパケットが同じ 1 つのキュー(CPU Queue)で処理されるようになっている。Distributed タイプの場合は、SEND 処理対象のパケットは CPU Queue で処理され、FORWARD／DELIVER 処理対象のパケットはそれぞれネットワークインタフェース毎に用意された個別のキュー(Input Queue)で処理されるようになっている。また、バックプレーン容量を無制限に設定することも可能(デフォルトは無制限)であり、この場合にはパケットはバックプレーンキューに格納されずに遅延なしで処理されることになる。

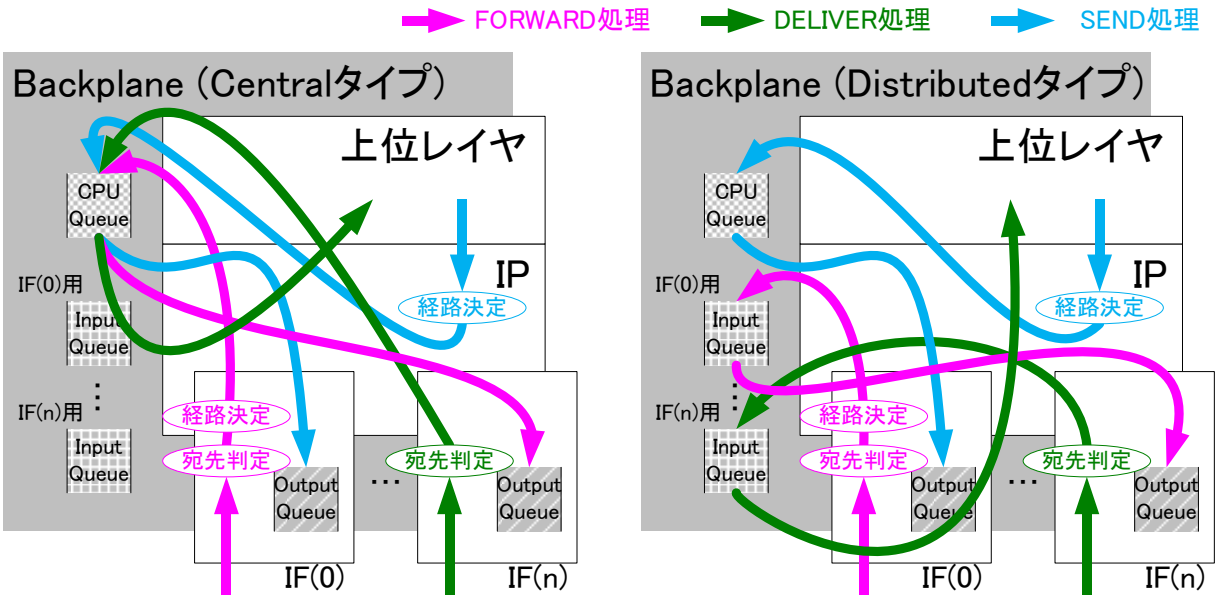

図 4-2 ルータバックプレーン処理概要

以下では、ルータバックプレーン処理の詳細を実装コードを見ながら解説する。

ルータバックプレーン処理は、以下の 3 ステップに分かれる。

1. バックプレーンキューへのパケットの Enqueue
2. パケット Dequeue イベントの登録
3. バックプレーンキューからのパケットの Dequeue

4.2.1.1.1 バックプレーンキューへのパケットの Enqueue

バックプレーンキューへのパケットの Enqueue は、NetworkIpSendOnBackplane 関数内で行われる。CPU Queue へ Enqueue する際には NetworkIpCpuQueueInsert 関数を呼び出し、各ネットワークインタフェース用に用意された Input Queue へ Enqueue する際には NetworkIpInputQueueInsert 関数を呼び出す。

```
14954 void //inline//
14955 NetworkIpSendOnBackplane(
14956       Node *node,
14957       Message *msg,
14958       int incomingInterface,
14959       int outgoingInterface,
14960       NodeAddress hopAddr)
14961 {

.... 中略 ....

14969     if (ip->backplaneThroughputCapacity ==
14970         NETWORK_IP_UNLIMITED_BACKPLANE_THROUGHPUT)
14971     {

.... 中略 ....

15003     }
15004     else
15005     {
15006         BOOL queueIsFull = FALSE;
15007
15008         if (incomingInterface == CPU_INTERFACE ||
15009             ip->backplaneType == BACKPLANE_TYPE_CENTRAL)
15010         {
15011             NetworkIpCpuQueueInsert(node,
15012                                     msg,
15013                                     hopAddr,
15014                                     ipHeader->ip_dst,
15015                                     outgoingInterface,
15016                                     NETWORK_PROTOCOL_IP,
15017                                     &queueIsFull,
15018                                     incomingInterface);
15019         }
15020         else
15021         {
15022             NetworkIpInputQueueInsert(node,
15023                                     incomingInterface,
15024                                     msg,
15025                                     hopAddr,
15026                                     ipHeader->ip_dst,
15027                                     outgoingInterface,
15028                                     NETWORK_PROTOCOL_IP,
15029                                     &queueIsFull);
15030         }
```

パケットの Enaueue

行番号	処理内容
15011 -15018	SEND 処理の場合、または FORWARD 処理でバックプレーンタイプが Central の場合の処理。パケットを CPU Queue へ Enqueue する。
15022 -15029	FORWARD 処理でバックプレーンタイプが Distributed の場合の処理。 incoming I/F Index に応じて、対応する Input Queue へ Enqueue する。

4.2.1.1.2 パケット Dequeue イベントの登録

パケット Dequeue イベントの登録は、NetworkIpSendOnBackplane 関数内でパケットの Enqueue 処理を行った直後に NetworkIpUseBackplaneIfPossible 関数を呼び出すことで行われる。NetworkIpUseBackplaneIfPossible 関数では、当該キューの先頭パケットを見て、そのサイズに応じた遅延時間後に発火するタイマッセージを作成し、QualNet のイベントスケジューラに登録する。遅延時間は以下のように計算される。

network_ip.cpp

```
14775         if (incomingInterface == CPU_INTERFACE
14776             || ip->backplaneType == BACKPLANE_TYPE_CENTRAL)
14777         {
14778             backplaneDelay = (clocktype) (packetSize * 8 * SECOND /
14779                                           ip->backplaneThroughputCapacity);
14780         }
14781         else
14782         {
14783             backplaneDelay = (clocktype) (packetSize * 8 * SECOND /
14784                 ip->interfaceInfo[incomingInterface]->disBackplaneCapacity);
14785         }
```

Dequeue イベントの登録

行番号	処理内容
14775 -14780	CPU Queue に入っているパケットの場合(SEND 処理の場合もしくはバックプレーンタイプが CENTRAL の場合)、ノードの IP レイヤに設定されているバックプレーン容量、すなわち ip->backplaneThroughputCapacity がその処理能力となる。packetSize は[byte]、処理能力は[bps]として遅延時間が計算される。
14781 -14785	Input Queue に入っているパケットの場合（DELIVER 処理または FORWARD 処理でバックプレーンタイプが Distributed の場合）、受信インタフェースに設定されているバックプレーン容量、すなわち ip->interfaceInfo[incomingInterface]->disBackplaneCapacity がその処理能力となる。

尚、各キューにおいて、パケットは当然逐次処理される。つまり、先頭パケットの遅延タイマがセットされている期間(すなわち、パケットがバックプレーンに於いて処理されているとみなされている期間)は後続のパケットは待たされる。先頭パケットの処理が終わりキューから Dequeue され次第、再び遅延タイマセット処理が行われる。その管理は backplaneStatus で行われ、下記は処理の一例である。

network_ip.cpp

```
14723     // If the interface is busy sending on the backplane, then wait...
14724     if (*backplaneStatus != NETWORK_IP_BACKPLANE_IDLE)
14725     {
14726        return;
14727     }
```

また、遅延タイマメッセージは下記の通り生成される。宛先は自身のネットワーク層で、メッセージ種別は MSG_NETWORK_Backplane である。

network_ip.cpp

```
14796             backplaneMsg = MESSAGE_Alloc(node,
14797                                          NETWORK_LAYER,
14798                                          NETWORK_PROTOCOL_IP,
14799                                          MSG_NETWORK_Backplane);
```

4.2.1.1.3 バックプレーンキューからのパケットの Dequeue

Dequeue イベントタイマは、通常の QualNet の IP レイヤのタイマイベントとして発生（NetworkIpLayer 関数で処理）し、NetworkIpReceiveFromBackplane 関数が呼び出され、その中で Dequeue 処理が行われる。

network_ip.cpp

```
2502 void
2503 NetworkIpLayer(Node *node, Message *msg)
2504 {
```

```
2505     switch (msg->protocolType)
2506     {

.... 中略 ....

2522         case NETWORK_PROTOCOL_IP:
2523         {
2524             switch (msg->eventType)
2525             {
2526                 // STATS DB CODE
2527
2528                 case MSG_NETWORK_Backplane:
2529                 {
2530                     NetworkIpReceiveFromBackplane(node, msg);
2531                     MESSAGE_Free(node, msg);
2532                     break;
2533                 }
```

Dequeue イベントの発生

行番号	処理内容
2530	NetworkIpReceiveFromBackplane 関数が呼び出される。

Dequeue イベントタイマの発火は、ルータバックプレーンにおけるパケットの処理完了を意味し、当該パケットは Input Queue（もしくは CPU Queue）から、その宛先に応じた場所へ配送される。

network_ip.cpp

```
14828 void
14829 NetworkIpReceiveFromBackplane(Node *node, Message *msg)
14830 {
.... 中略 ....

14841     NetworkIpBackplaneInfo *info = (NetworkIpBackplaneInfo *)
14842                                     MESSAGE_ReturnInfo(msg);
14843
14844     if (info->incomingInterface == CPU_INTERFACE
14845         || ip->backplaneType == BACKPLANE_TYPE_CENTRAL)
14846     {
14847         backplaneStatus = &ip->backplaneStatus;
14848
14849         NetworkIpCpuQueueDequeuePacket(node,
14850                                        &queueMsg,
14851                                        &hopAddr,
14852                                        &nextHopMacAddr,
14853                                        &outgoingInterface,
14854                                        &networkType,
14855                                        &priority);
14856     }
14857     else
14858     {
14859         backplaneStatus =
14860                   &ip->interfaceInfo[info->incomingInterface]-
>backplaneStatus;
14861
14862         NetworkIpInputQueueDequeuePacket(node,
14863                                        info->incomingInterface,
14864                                        &queueMsg,
14865                                        &hopAddr,
14866                                        &nextHopMacAddr,
14867                                        &outgoingInterface,
14868                                        &networkType,
14869                                        &priority);
14870     }
```

処理完了パケットのデキュー

行番号	処理内容
14841 -14842	Dequeue すべきキューの情報を取り出す。これは遅延タイマに送信時に付与しておいた Info である。
14844 -14870	Enqueue の時と同様、incoming I/F Index に応じて、対応する Input Queue または CPU Queue から Dequeue される。ただしバックプレーンタイプが Central の場合は、必ず CPU Queue から Dequeue される。

4.2.1.2 設定パラメータ反映処理

ルータモデルの設定パラメータのうち、ROUTER-BACKPLANE-TYPE, ROUTER-BACKPLANE-THROUGHPUT, ROUTER-PERFORMANCE-VARIATION のパラメータがどのように読み込まれるかについて以下で説明する。

network_ip.cpp

```
1311     IO_ReadString(
1312         node->nodeId,
1313         ANY_ADDRESS,
1314         nodeInput,
1315         "ROUTER-BACKPLANE-THROUGHPUT",
1316         &retVal,
1317         backplaneThroughput);
1318
1319     char *p;
1320
1321     if (retVal == FALSE ||
1322         strcmp(backplaneThroughput, "UNLIMITED") == 0 ||
1323         strcmp(backplaneThroughput, "0") == 0)
1324     {
1325         // If not specified, we assume infinite backplane throughput.
1326         ip->backplaneThroughputCapacity =
1327             NETWORK_IP_UNLIMITED_BACKPLANE_THROUGHPUT;
1328     }
```

ROUTER-BACKPLANE-THROUGHPUT

行番号	処理内容
1311	まず、ノードパラメタとして ROUTER-BACKPLANE-THROUGHPUT を読み込む。
1321-1326	指定されていないか、“UNLIMITED” か”0”が指定されている場合は、無限大のスループット(ルータバックプレーンにおける処理遅延なし)とみなされる。

network_ip.cpp

```
1329     else if ((throughput = strtod(backplaneThroughput , &p)) > 0.0)
1330     {
1331         IO_ReadFloat(
1332             node->nodeId,
1333             ANY_ADDRESS,
1334             nodeInput,
1335             "ROUTER-PERFORMANCE-VARIATION",
1336             &retVal,
1337             &rtPerformVar);
1338
1339         if (retVal)
1340         {
1341             throughput += ((throughput * rtPerformVar) / 100);
1342         }
1343
```

ROUTER-PERFORMANCE-VARIATION

行番号	処理内容
1329	ROUTER-BACKPLANE-THROUGHPUT で正の値が指定されている場合のみ、throughput にその値を格納し、続く処理を行う。
1331-1341	ROUTER-PERFORMANCE-VARIATION が指定されていた場合、throughput の値を ROUTER-PERFORMANCE-VARIATION [%]増しとする。

network_ip.cpp

```
1344          ip->backplaneThroughputCapacity = (clocktype)throughput;
1345
1346          for (i = 0; i < node->numberInterfaces; i++)
1347          {
1348              ip->interfaceInfo[i]->disBackplaneCapacity =
1349                  (clocktype)(throughput / (node->numberInterfaces + 1));
1350          }
1351
```

各フィールドへのセット

行番号	処理内容
1344	ノードの処理能力に相当する ip->backplaneThroughputCapacity へ throughput を格納する。
1346-1350	I/F 毎の処理能力に相当する ip->interfaceInfo[i]->disBackplaneCapacity へ throughput / (node->numberInterfaces + 1)を格納する。分母が”+1”となっているのは CPU を含むためである。全 I/F と CPU が全体として指定の処理能力を保持していると考える。ちなみに、ここで ip->interfaceInfo[i]->disBackplaneCapacity の値が無条件で設定されているが、実際にはタイプが DISTRIBUTED の場合にしか用いられない。

network_ip.cpp

```
1352          IO_ReadString(
1353              node->nodeId,
1354              ANY_ADDRESS,
1355              nodeInput,
1356              "ROUTER-BACKPLANE-TYPE",
1357              &retVal,
1358              rtBackType);
1359
1360          if (strcmp(rtBackType, "CENTRAL") == 0)
1361          {
1362              ip->backplaneType = BACKPLANE_TYPE_CENTRAL;
1363          }
1364          else if (strcmp(rtBackType, "DISTRIBUTED"))
1365          {
1366              ERROR_ReportError("ROUTER-BACKPLANE-TYPE should be "
1367                                "either \"CENTRAL\" or \"DISTRIBUTED\".\n");
1368          }
1369          else
1370          {
1371              ip->backplaneThroughputCapacity =
1372                  (clocktype)(throughput / (node->numberInterfaces + 1));
1373          }
```

ROUTER-BACKPLANE-TYPE

行番号	処理内容
1352-1368	ROUTER-BACKPLANE-TYPE には"CENTRAL"か"DISTRIBUTED"のどちらかが指定され、デフォルトは"DISTRIBUTED"である。それ以外を指定した場合はエラーとなる。
1369-1373	"DISTRIBUTED"が指定されていた場合は、ip->backplaneThroughputCapacity へ throughput / (node->numberInterfaces + 1)を格納する。これにより、ip->backplaneThroughputCapacity には CENTRAL の場合は、指定された throughput が、DISTRIBUTED の場合は、I/F を含めた処理ユニットで分配された値がセットされることになる。

上記のまとめとして、以下に設定例を示す。

- 設定例 1

```
ROUTER-BACKPLANE-TYPE                   DISTRIBUTED
ROUTER-BACKPLANE-THROUGHPUT             60000000
ROUTER-PERFORMANCE-VARIATION            5
インタフェース数:                        2 枚
```

フィールド	値
ip->backplaneType	DISTRIBUTED
ip->backplaneThroughputCapacity	21,000,000 (= 60,000,000 * 1.05 / (2 + 1)) [bps]
ip->interfaceInfo[i]	21,000,000 (= 60,000,000 * 1.05 / (2 + 1)) [bps]

- 設定例 2

```
ROUTER-BACKPLANE-TYPE                   CENTRAL
ROUTER-BACKPLANE-THROUGHPUT             100000000
ROUTER-PERFORMANCE-VARIATION            0
インタフェース数:                        3 枚
```

フィールド	値
ip->backplaneType	DISTRIBUTED
ip->backplaneThroughputCapacity	100,000,000 [bps]
ip->interfaceInfo[i]	25,000,000 (= 100,000,000 / (3 + 1)) [bps] (ただしこのフィールドは参照されない)

4.3 IP パケットプロセッシング（送信・受信・転送処理）

本節では、IP レイヤにおいて送受信パケットがどのようにハンドリングされるのかについて、QualNet の IP レイヤの実装コードを追いながら詳細に解説する。ただし QualNet の実装コードを追う前に、まずは一般の IP 転送処理の流れについて理解を深めるために、Linux カーネル 2.4 における IP レイヤでのパケット処理を眺めてみることにする。

4.3.1 Linux カーネル 2.4 における IP パケットプロセッシング概観

QualNet の解説に入る前に、まずそもそもの IP レイヤ処理について理解するために、Linux カーネル 2.4 における IP 処理の実装について簡単に触れておく。図 4-3 は、Linux カーネル 2.4 における IP パケットプロセッシングの実装イメージである。（詳細は、『Linux V2.4 カーネル内部解析報告 ドラフト 第 4 版』[3] を参照。）図中の括弧( )付きの文字列はカーネル内の処理関数名を表しており、sk_buff はパケットを格納しているメモリ領域（QualNet の Message インスタンスのようなもの）を表している。角を丸めた四角形は経路表（ルーティングテーブル）ならびにアドレス解決表（ARP テーブル）であり、それぞれ経路情報（宛先 IP アドレスと outgoing インタフェース／次ホップ IP アドレスのマッピング情報）とアドレス解決情報（IP アドレスと MAC アドレスのマッピング情報）を格納したメモリ領域を表している。

図では経路表が 2 つ描かれており、ip_route_input( )と ip_route_output( )がそれぞれ別の経路表を使っているように見えるが、たいていの場合は同一の経路表が使われる。（受信インタフェースと送信インタフェースで異なる経路制御プロトコルが動作し、かつ経路制御プロトコルが独自の経路表を用いているような場合には、異なる経路表が用いられることもある。）

[3] http://sourceforge.jp/projects/linux-kernel-docs/wiki/internal24-index

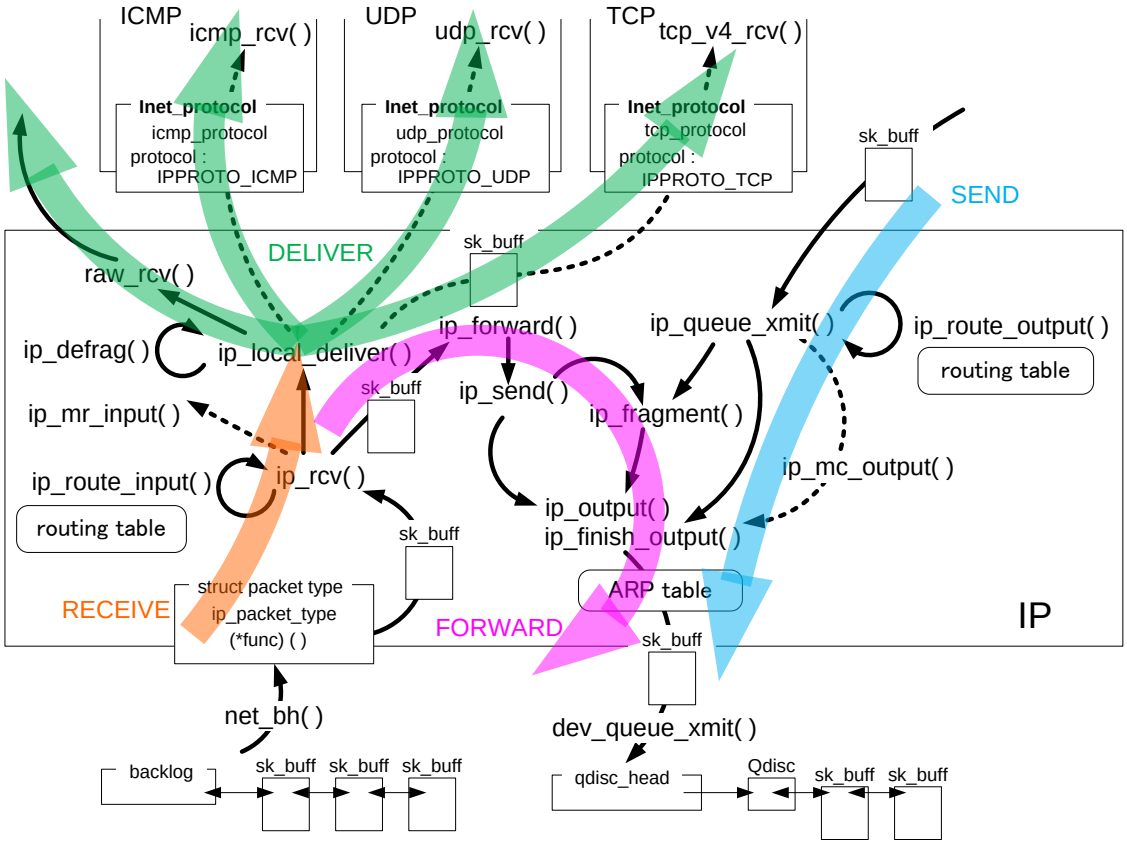

図 4-3 Linux カーネル 2.4 における IP 転送処理

IP パケットプロセッシングには、大きく分けて、上位レイヤから渡されたパケットの処理（SEND 処理）と下位レイヤから渡されたパケットの処理（RECEIVE 処理）の 2 種類がある。さらに RECEIVE 処理は、下位レイヤから受け取ったパケットをそのまま自ノードで処理する DELIVER 処理と、他ノードに転送する FORWARD 処理に分岐する。

まずは SEND 処理について簡単に眺めてみることにする。SEND 処理では、上位レイヤからパケットを受け取るとまず ip_queue_xmit( )関数が呼ばれ、その中で ip_route_output( )関数を呼び出して経路（outgoing インタフェースと nexthop）の決定を行う。経路が決定すると、ip_queue_xmit( )関数は IP ヘッダを付与し、必要に応じて ip_fragment( )関数を呼び出して IP パケットのフラグメント化を行い、ip_output( )関数（マルチキャストパケットの場合は ip_mc_output( )関数）を呼び出す。これらの関数から ip_finish_output( )関数が呼び出されてその中で ARP 解決を行い、宛先 MAC アドレスが判明した時点で dev_queue_xmit( )関数を呼び出してドライバを起動し、その後のパケット処理をネットワークデバイスドライバに引き継ぐ。SEND 処理の概要は以上である。

次に、RECEIVE 処理について眺めてみる。まず、ネットワークデバイスドライバでの受信処理が終わると ip_rcv( )関数が呼び出される。ここが IP レイヤにおける RECEIVE 処理の始まりである。ip_rcv( )関数は IP ヘッダの整合性チェックを行った後、ip_route_input( )関数を呼び出して経路の決定（自ノード宛か他ノード宛かのチェックと他ノード宛の場合に nexthop と outgoing インタフェースの決定）を行う。ここで、他ノード宛のパケットの場合には FORWARD 処理に分岐し、ip_forward( )関数、さらに ip_send( )関数が呼ばれてその先は SEND 処理の場合と同様の処理が行われる。他ノード宛ではなく自ノード宛のパケットの場合には DELIVER 処理に分岐し、ip_local_deliver( )関数（マルチキャストパケットの場合は ip_mr_input( )関数）が呼ばれて IP ヘッダのプロトコル ID フィールドに対応する上位レイヤプロトコルに

パケット処理を引き継ぐ。なお、フラグメント化された IP パケットに対しては、ip_local_deliver( )関数は ip_defrag( )関数を呼び出してリアセンブル処理を行い、全てのフラグメントが到着してリアセンブルが完了した時点で上位レイヤへのパケット処理の引き継ぎを行う。

以上が Linux カーネル 2.4 における IP パケットプロセッシングの流れである。以後、QualNet における IP パケットプロセッシングについて解説するが、基本的にはこの処理の流れを踏襲していることが確認できるかと思う。

4.3.2 QualNet における IP パケットプロセッシング概観

QualNet における実装に関しても、先ほどと同じように SEND 処理と RECEIVE 処理(さらに FORWARD 処理と DELIVER 処理)を順に追っていくことにする。QualNet の実装では、SEND 処理、RECEIVE 処理ともに前述のバックプレーン処理を挟んだ前半処理と後半処理に分かれる。前半処理は経路選択、後半処理は上下位レイヤへのパケット受け渡しが主な処理内容である。

4.3.2.1 SEND 処理

図 4-4 に QualNet における SEND 処理の実装イメージを示す。

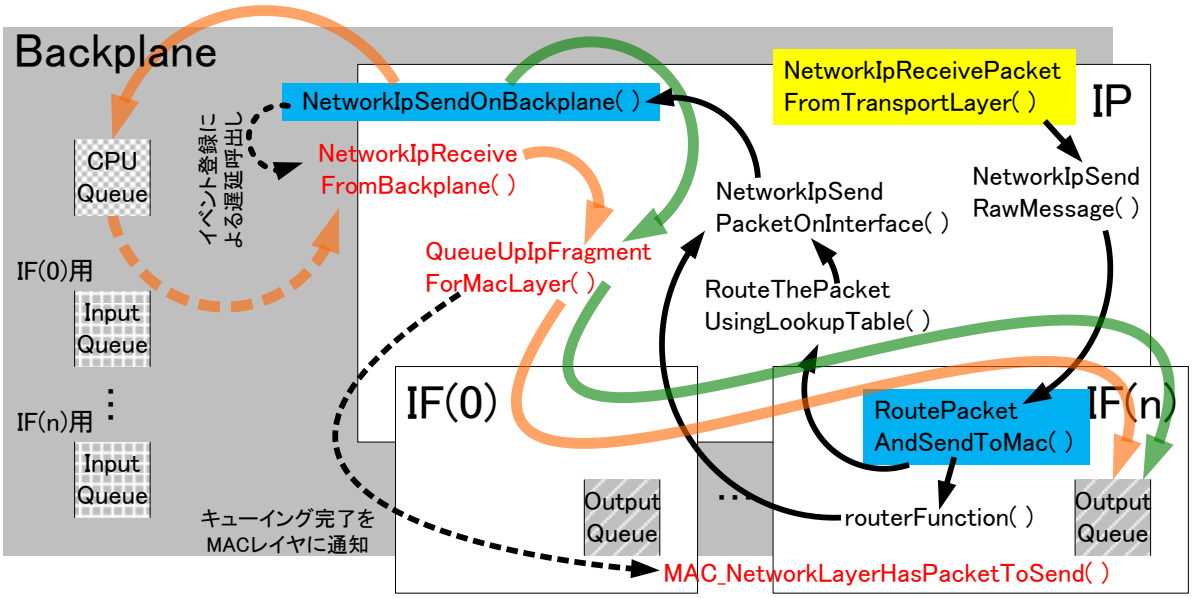

図 4-4 QualNet における SEND 処理

SEND 処理は、まず NetworkIpReceivePaketFromTransportLayer( )関数が上位レイヤから呼ばれるところから始まる。NetworkIpReceivePacketFromTransportLayer( )関数は、NetworkIpSendRawMessage( )関数を呼び出すだけの関数である。NetworkIpSendRawMessage( )関数は、outgoing インタフェースがまだ決定していなければ送信元 IP アドレスから outgoing インタフェースを決定し、経路決定とパケット送信を行うために RoutePacketAndSendToMac( )関数を呼び出す。

RoutePacketAndSendToMac( )関数は、ルーティングプロトコル独自の経路決定(選択)関数である routerFunction( )関数、または経路表を用いた経路選択関数である RouteThePacketUsingLookupTable( )関数を呼び出す。これらの関数は、各々決められた方法により当該パケットに対する経路を決定し、NetworkIpSendPacketOnInterface( )関数を呼び出す。NetworkIpSendPacketOnInterface( )関数は、NetworkIpSendOnBackplane( )関数を呼び出して、ルータバックプレーン処理にパケット処理を委ねる。

ルータバックプレーン処理（NetworkIpReceiveFromBackplane( )関数の遅延呼び出し処理）を経たパケットは、QueueUpIpFragmentForMacLayer( )関数で処理を受ける。QueueUpIpFragmentForMacLayer( )関数は、当該パケットを outgoing インタフェースの Output キューに Enqueue した後、MAC 層 API である MAC_NetworkLayerHasPacketToSend( )関数を呼び出して、outgoing インタフェースの MAC プロトコルに対して「Network 層から送るべきパケットがあります」という通知を行い、MAC 層に処理を委ねる。

4.3.2.2 RECEIVE 処理

RECEIVE 処理は、まず NetworkIpReceivePacket( )関数が下位レイヤから呼ばれるところから始まる。NetworkIpReceivePacket( )関数は、IsMyPacket( )関数を呼び出して受信パケットが他ノード宛か自ノード宛かを判定する。他ノード宛の場合は、ForwardPacket( )関数を呼び出して FORWARD 処理を実施する。自ノード宛の場合は、NetworkIpSendOnBackplane( )関数を呼び出して DELIVER 処理を実施する。

4.3.2.2.1 FORWARD 処理

図 4-5 に QualNet における FORWARD 処理の実装イメージを示す。

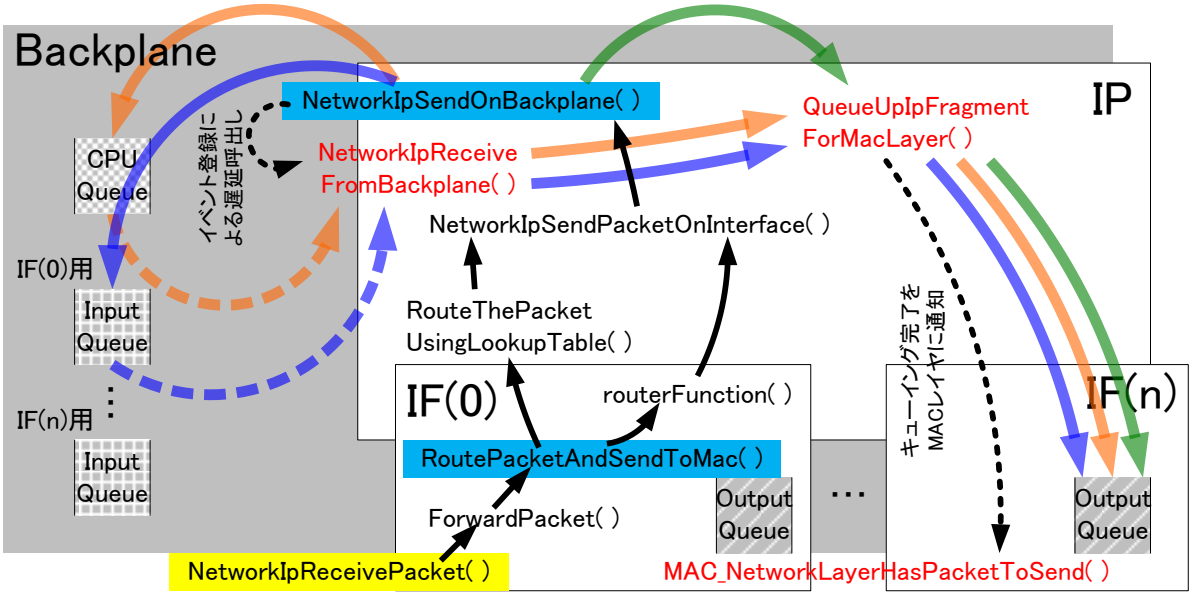

図 4-5 QualNet における FORWARD 処理

FORWARD 処理は、前半処理・後半処理ともに SEND 処理とほぼ同じである。唯一違うのは、前半処理が outgoing インタフェースに紐付けられて行われるのではなく、incoming インタフェースに紐付けられて行われる点である。よって routerFunction( )を呼び出した際には、outgoing インタフェース上で動作させている経路制御プロトコルではなく、incoming インタフェースで動作させている経路制御プロトコルの経路選択関数が呼び出される。

4.3.2.2.2 DELIVER 処理

図 4-6 に QualNet における DELIVER 処理の実装イメージを示す。

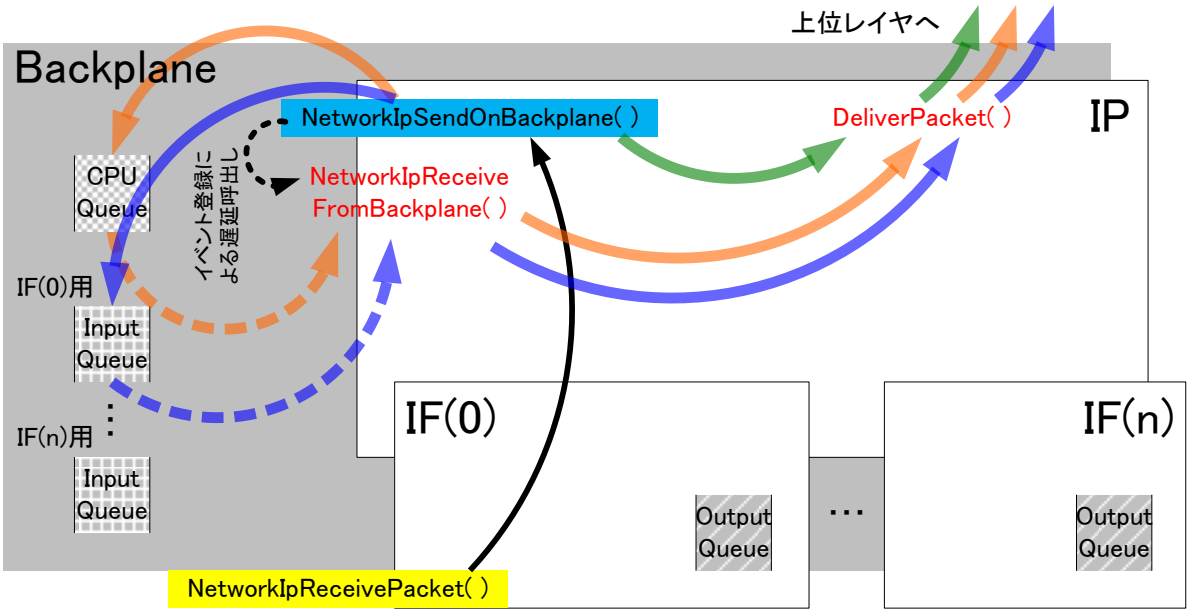

図 4-6 QualNet における DELIVER 処理

前半処理は、NetworkIpSendOnBackplane( )関数を呼び出すだけである。NetworkIpSendOnBackplane( )関数では、バックプレーン容量が無制限でなければ、CPU Queue あるいは Input Queue に受信パケットを Enqueue したうえで、バックプレーン容量に見合った遅延時間後に後半処理が行われるようにイベント登録を行う。バックプレーン容量が無制限の場合は、直接 DeliverPacket( )関数を呼び出して、引き続き後半処理を実施する。

後半処理は、DeliverPacket( )関数で実施される。DeliverPacket( )関数では、IP ヘッダのプロトコル番号をチェックし、対応する上位プロトコルの受信関数にパケットを引き渡す。

4.4 経路表

本節では、QualNet における経路表のデータ構造と経路エントリの追加・削除 API について、実装コードを追いながら解説する。

4.4.1 経路表の構造

Forwarding Table 及びそのテーブルエントリは、構造体として定義されている。以下に構造体の定義部を示す。

network_ip.h

```
2135 //---------------------------------------------------------
2136 // Routing table (forwarding table)
2137 //---------------------------------------------------------
2138
2139 // /**
2140 // STRUCT      :: NetworkForwardingTableRow
2141 // DESCRIPTION :: Structure of an entity of forwarding table.
2142 // **/
2143 typedef
2144 struct
2145 {
2146     NodeAddress destAddress;        // destination address
2147     NodeAddress destAddressMask;    // subnet destination Mask
2148     int interfaceIndex;           // index of outgoing interface
2149     NodeAddress nextHopAddress;     // next hop IP address
2150
2151     int cost;
2152
2153     // routing protocol type
2154     NetworkRoutingProtocolType protocolType;
2155
2156     // administrative distance for the routing protocol
2157     NetworkRoutingAdminDistanceType adminDistance;
2158
2159     BOOL interfaceIsEnabled;
2160 }
2161 NetworkForwardingTableRow;
2162
2163 // /**
2164 // STRUCT      :: NetworkForwardingTable
2165 // DESCRIPTION :: Structure of forwarding table.
2166 // **/
2167 typedef
2168 struct
2169 {
2170     int size;                       // number of entries
2171     int allocatedSize;
2172 #ifdef ADDON_STATS_MANAGER
2173     D_String *tableStr;
2174 #endif
2175     NetworkForwardingTableRow *row;  // allocation in Init function in Ip
2176 }
2177 NetworkForwardingTable;
```

NetworkForwardingTable が、Fowarding Table 全体を表す構造体であり、NetworkForwardingTableRow がテーブルの 1 エントリ（宛先アドレスとネクストホップの 1 つのペア）を表している。

表 4-2 に、NetworkForwardingTableRow 構造体の各メンバの説明を示す。

表 4-2 Forwarding Table エントリ(NetworkForwardingTableRow 構造体)のメンバ

メンバ	説明
destAddress	宛先アドレス
destAddressMask	宛先アドレスのネットマスク
interfaceIndex	宛先へ送信するためのタンターフェイス番号(0～)
nextHopAddress	ネクストホップアドレス
cost	この経路のメトリック値. NetworkUpdateForwardingTable()関数で Forwarding Table を更新する際に指定される。いくつかのルーティングプロトコルにより使用される。
protocolType	当該エントリを追加/更新した Routing Protocol の種類が格納される. (ROUTING_PROTOCOL_STATIC, ROUTING_PROTOCOL_ICMP_REDIRECT, ROUTING_PROTOCOL_BELLMANFORD etc...)
adminDistance	Administrative Distance(AD). 複数のルーティングプロトコルにより同じネットワーク宛ての異なる NextHop のエントリが登録された場合、どのプロトコロルの結果を優先するかを決める値(信頼度). この値が小さいほど優先度(信頼度)が高いことを表す。NetworkRoutingGetAdminDistance 関数により Routing Protocol Type をキーに取得される。同じ宛先アドレス/サブネットのエントリは AD 値が小さい順に格納されている。 なお、CISCO の AD 値の説明は下記 URL で見ることができる。QualNet では CISCO の推奨する値とは厳密には異なる点に留意. .http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094195.shtml
interfaceIsEnabled	interfaceIndex で示されるインタフェースが有効かどうかを格納している.

Forwarding Table 内の各エントリは以下の優先順位でソートされた状態で格納されている。

1. 宛先アドレスについて降順
2. 宛先アドレスのネットマスクについて降順
3. AD について昇順
4. cost について昇順(※ただしこの点に関しては、ソースコードにバグがあるようである)

ここでアドレスの順序は、アドレスを符号なし整数値としてみた場合の順序と同じである。図 4-7 に格納例を示す。

```
-------------------------------------------------------------------
| #   | DstAddress    | Dst Netmask   | NextHop       | AD |
-------------------------------------------------------------------
| 0   | 241.3.200.1   | 255.0.0.0     | 240.3.20.5    | 101 |
| 1   | 192.168.0.0   | 255.255.255.0 | 100.2.20.1    | 150 |
| 2   | 192.168.0.0   | 255.255.255.0 | 100.2.20.1    |   1 |
| 3   | 192.168.0.0   | 255.255.255.0 | 100.1.10.1    |   0 |
| 4   | 10.41.50.0    | 255.255.255.0 | 10.41.0.0     |   1 |
| 5   | 10.41.50.0    | 255.255.0.0   | 10.41.0.0     | 200 |
```

図 4-7 Forwarding Table の格納順

上記は、ユニキャスト(単一の Next-Hop)の場合の Forwarding Table であり、マルチキャスト向けの Forwarding Table は別に定義されている。

以下に、マルチキャスト向け Forwarding Table 及びエントリの構造体定義を示す。

network_ip.h

```
1865 // /**
1866 // STRUCT      :: NetworkMulticastForwardingTableRow
1867 // DESCRIPTION :: Structure of an entity of multicast forwarding table.
1868 // **/
1869 typedef
1870 struct
1871 {
1872     NodeAddress sourceAddress;
1873     NodeAddress sourceAddressMask;        // Not used
1874     NodeAddress multicastGroupAddress;
1875     LinkedList *outInterfaceList;
1876 } NetworkMulticastForwardingTableRow;
1877
1878 // /**
1879 // STRUCT      :: NetworkMulticastForwardingTable
1880 // DESCRIPTION :: Structure of multicast forwarding table
1881 // **/
1882 typedef
1883 struct
1884 {
1885     int size;
1886     int allocatedSize;
1887     NetworkMulticastForwardingTableRow *row;
1888 } NetworkMulticastForwardingTable;
```

ユニキャストの場合との違いは、Next-Hop アドレスが、マルチキャストアドレス(multicastGroupAddress)になっている点と、パケットの送出先インタフェースが複数指定可能なようリスト (outInterfaceList)になっている点である。なお、QualNet 5.2 ではマルチキャストの Forwarding Table はスタティックルートの場合しか用いられていない。

4.4.2 経路エントリの追加・削除

Forwarding Table の操作に関連する関数一覧及び概要を表 4-3 に示す。これらの関数は、network_ip.h, cpp に定義されている。

表 4-3 Forwarding Table 操作関連の関数

関数	概要
NetworkInitForwardingTable	Forwarding Table の初期化
NetworkUpdateForwardingTable	Forwarding Table へのエントリの追加またはエントリの変更を行う
NetworkRemoveForwardingTableEntry	Forwarding Table からエントリの削除を行う
NetworkEmptyForwardingTable	Forwarding Table を空にする(全て削除)
NetworkPrintForwardingTable	Forwarding Table の内容を標準出力に表示する

本節では、この内、Forwarding Table へのエントリの追加関数(NetworkUpdateForwardingTableEntry)及び削除関数(NetworkRemoveForwardingTableEntry)の 2 つについて説明する。

4.4.2.1 エントリの追加

エントリの追加は、NetworkUpdateForwardingTable 関数で実施される。追加処理の流れを図 4-8 に示す。

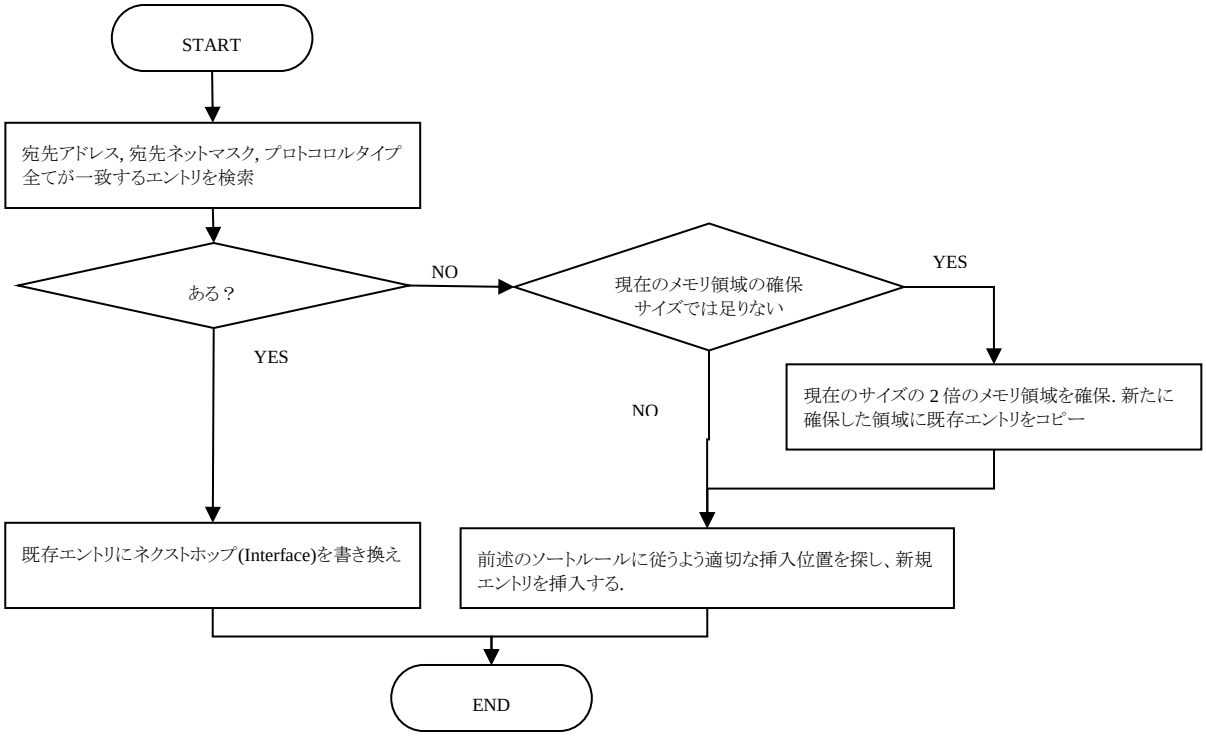

図 4-8 エントリの追加処理の流れ

まず、既存の Forwarding Table のエントリの中から引数で指定された宛先アドレス、サブネットマスク、及び protocolType の全てが一致するエントリを検索し、一致するエントリがあれば情報を書き換え、なければ新たにエントリを作成するようになっている。エントリを新たに作成する際には、メモリアロケーションを頻繁に行わないで済むよう工夫が行われている。

以下ソースコードを見ながら、詳細な説明を行う。

network_ip.cpp

```
11220 void
11221 NetworkUpdateForwardingTable(
11222     Node *node,
11223     NodeAddress destAddress,
11224     NodeAddress destAddressMask,
11225     NodeAddress nextHopAddress,
11226     int interfaceIndex,
11227     int cost,
11228     NetworkRoutingProtocolType type)
11229 {
11230     NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
11231     NetworkForwardingTable *forwardTable = &(ip->forwardTable);
11232
11233     NetworkRoutingProtocolType newType = type;
11234     NetworkRoutingAdminDistanceType adminDistance;
11235     int i;
11236
11237     //loop through to find the first entry where destination address,
11238     //mask, and type all match.  For ICMP redirect messages, type is
11239     //obtained from the forwarding table and doesn't need to match.
11240     for (i = 0; i < forwardTable->size
11241         && (forwardTable->row[i].destAddress != destAddress
```

```
11242             || forwardTable->row[i].destAddressMask != destAddressMask
11243             || (forwardTable->row[i].protocolType != type
11244             && type != ROUTING_PROTOCOL_ICMP_REDIRECT)); i++)
11245     {
11246         //loop until match
11247     }
```

更新するエントリの検索

行番号	処理内容
11240-11247	このループでは、既存のエントリの中から宛先アドレス、サブネットマスク及び protocolType の全てが一致するエントリが検索され、そのインデックスが i に格納される。見つからなかった場合は、i は fowardTable->size に等しい点に留意。

network_ip.cpp

```
11248
11249     if ((type == ROUTING_PROTOCOL_ICMP_REDIRECT) && (i == forwardTable-
>size))
11250     {
11251         newType = ROUTING_PROTOCOL_DEFAULT;
11252         adminDistance = ROUTING_ADMIN_DISTANCE_DEFAULT;
11253     }
11254     else if ((type == ROUTING_PROTOCOL_ICMP_REDIRECT) && (i != forwardTable-
>size))
11255     {
11256         //admin distance and type are retrieved from forwarding table
11257         adminDistance = forwardTable->row[i].adminDistance;
11258         newType = forwardTable->row[i].protocolType;
11259     }
11260     else
11261     {
11262         //adminDistance is obtained from the type
11263         adminDistance = NetworkRoutingGetAdminDistance(node, type);
11264     }
```

AD値の取得

行番号	処理内容
11263	ルーティングプロトコルをキーに AD 値を取得。なお、ノードも引数にわたっているが、基本的には使用していないと考えて良い。(一部の特殊なライブラリで使用)

```
11265
11266     NetworkRouteUpdateEventType routeUpdateFunction = NULL;
11267
11268     if (interfaceIndex == ANY_INTERFACE)
11269     {
11270         if (nextHopAddress == (unsigned) NETWORK_UNREACHABLE)
11271         {
11272             interfaceIndex = DEFAULT_INTERFACE;
11273         }
11274         else
11275         {
11276             interfaceIndex = NetworkIpGetInterfaceIndexForNextHop(
11277                 node,
11278                 nextHopAddress);
11279         }
11280     }
11281
11282     if (interfaceIndex < 0)
11283     {
11284         char err[MAX_STRING_LENGTH];
11285         char addr[MAX_STRING_LENGTH];
11286
11287         IO_ConvertIpAddressToString(nextHopAddress, addr);
11288         sprintf(err, "Node %u: Next hop %s is not connected to this node\n",
```

```
11289             node->nodeId, addr);
11290         ERROR_ReportError(err);
11291     }
11292
```

インタフェース番号の取得

行番号	処理内容
11276-11291	NextHop アドレスからインタフェース番号を取得している。NextHop に到達可能なインタフェースが見つからなかった場合(interfaceIndex < 0)はエラーを出力している。

```
11307
11308     if (i == forwardTable->size)
11309     {
11310         forwardTable->size++;
11311
11312         if (forwardTable->size > forwardTable->allocatedSize)
11313         {
11314             if (forwardTable->allocatedSize == 0)
11315             {
11316                 forwardTable->allocatedSize =
FORWARDING_TABLE_ROW_START_SIZE;
11317                 forwardTable->row = (NetworkForwardingTableRow*)
11318                     MEM_malloc(
11319                         forwardTable->allocatedSize *
11320                         sizeof(NetworkForwardingTableRow));
11321             }
11322             else
11323             {
11324                 int newSize = (forwardTable->allocatedSize * 2);
11325
11326                 NetworkForwardingTableRow* newTableRow =
11327                     (NetworkForwardingTableRow*)MEM_malloc(
11328                         newSize * sizeof(NetworkForwardingTableRow));
11329
11330                 memcpy(newTableRow, forwardTable->row,
11331                        (forwardTable->allocatedSize *
11332                         sizeof(NetworkForwardingTableRow)));
11333
11334                 MEM_free(forwardTable->row);
11335                 forwardTable->row = newTableRow;
11336                 forwardTable->allocatedSize = newSize;
11337             }//if//
11338         }//if//
11339
11340         while (i > 0 &&
11341                (destAddress > forwardTable->row[i - 1].destAddress
11342                 || (destAddress == forwardTable->row[i - 1].destAddress
11343                     && destAddressMask > forwardTable->row[i - 1].
11344                     destAddressMask)
11345                 || (destAddress == forwardTable->row[i - 1].destAddress
11346                     && destAddressMask == forwardTable->row[i - 1].
11347                     destAddressMask
11348                     &&  adminDistance < forwardTable->row[i -
1].adminDistance)
11349                 || (destAddress == forwardTable->row[i - 1].destAddress
11350                     && destAddressMask == forwardTable->row[i - 1].
11351                     destAddressMask
11352                     && cost == forwardTable->row[i - 1].cost
11353                     && cost < forwardTable->row[i - 1].cost)))
11354         {
11355             forwardTable->row[i] = forwardTable->row[i - 1];
11356             i--;
11357         }//while//
11358     }//if//
```

エントリの追加処理

行番号	処理内容
11308	この if が真となる場合は、一致するエントリが見つからなかった場合である。つまり、この if 文の中は新たに Forwarding Table にエントリを追加する処理となっている。if 文に入らない場合は、この後の処理で既存エントリの情報を上書きが行われるtこになる。
11312-11338	この if ブロックは、メモリ割当領域が足りなかった場合にさらに大きい領域を確保している処理である。最初は FORWARDING_TABLE_ROW_START_SIZE(=4)エントリ分の領域が確保され、それ以降足りなくなる毎に 8, 16, 32 ...と 2 倍ずつ確保されてゆく。
11340	11340 行目の while ブロックは、新規エントリを挿入する箇所を探索する処理である。新規エントリを挿入するスペースを空けるため、後のエントリを 1 個ずつ後方にずらしている。この処理により結果的に Forwarding Table の内容が 1.3.1 で示したソート順で保持されることになる。

```
11359
11360     forwardTable->row[i].destAddress = destAddress;
11361     forwardTable->row[i].destAddressMask = destAddressMask;
11362     forwardTable->row[i].interfaceIndex = interfaceIndex;
11363     forwardTable->row[i].nextHopAddress = nextHopAddress;
11364     forwardTable->row[i].protocolType = newType;
11365     forwardTable->row[i].adminDistance = adminDistance;
11366
11367     forwardTable->row[i].cost = cost;
11368
11369     if (NetworkIpInterfaceIsEnabled(node, interfaceIndex))
11370     {
11371         forwardTable->row[i].interfaceIsEnabled = TRUE;
11372     }
11373     else
11374     {
11375         forwardTable->row[i].interfaceIsEnabled = FALSE;
11376     }
11377
11378     routeUpdateFunction = NetworkIpGetRouteUpdateEventFunction(node);
11379
11380     if (routeUpdateFunction)
11381     {
11382         (routeUpdateFunction)(node, destAddress, destAddressMask,
11383                 nextHopAddress, interfaceIndex, cost, adminDistance);
11384     }
11385 }
```

送出先インタフェース番号の取得

行番号	処理内容
11369-11376	パケット送出先インタフェースの有効/無効を調べエントリの”interfaceIsEnabled”メンバに書き込みを行っている。なお、ここで無効となったエントリは、次回以降 Forwarding Table を参照して経路探索が行われる際、その検索対象から外されるようになっている。
11378	Forwarding Table の更新内容を通知するコールバック関数の呼び出し処理が行われている。現状、あまり使われていない。

4.4.2.2 エントリの削除

エントリの追加は、NetworkRemoveForwardingTableEntry 関数で実施される。この関数は、宛先アドレス/サブネットマスク, NextHop アドレス、送出先 Interface を引数に受け取る。

現在の Forwarding Table から、これら、宛先アドレス/サブネット, NextHop アドレス、送出 Interface の全てが一致するエントリを全て削除するようになっている。エントリの追加時と違い、ルーティングプロトコルの種類は見ていない点に注意しよう。

```
11400 void
11401 NetworkRemoveForwardingTableEntry(
11402     Node *node,
11403     NodeAddress destAddress,
11404     NodeAddress destAddressMask,
11405     NodeAddress nextHopAddress,
11406     int outgoingInterfaceIndex)
11407 {
11408     NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
11409     NetworkForwardingTable *rt = &ip->forwardTable;
11410
11411     int i = 0;
11412
11413     // Go through the routing table...
11414     while (i < rt->size)
11415     {
11416         // Delete entries that corresponds to the routing protocol used
11417         if ((rt->row[i].destAddress == destAddress) &&
11418             (rt->row[i].destAddressMask == destAddressMask) &&
11419             (rt->row[i].nextHopAddress == nextHopAddress) &&
11420             (rt->row[i].interfaceIndex == outgoingInterfaceIndex) )
11421         {
11422             int j = i + 1;
11423
11424             // Move all other entries down
11425             while (j < rt->size)
11426             {
11427                 rt->row[j - 1] = rt->row[j];
11428                 j++;
11429             }
11430
11431             // Update forwarding table size.
11432             rt->size--;
11433         }
11434         else
11435         {
11436             i++;
11437         }
11438     }
11439 }
11440
```

4.4.3 経路探索

Next Hop 探索は NetworkGetInterfaceAndNextHopFromForwardingTable 関数で行われる。ただし、同一名称で引数の異なる関数が 6 つ定義されていて、それぞれ少し探索方法が異なる。それぞれ表 4-4 のような動作となっている。

表 4-4 Next Hop 探索関数

#	関数定義	概要
1	void NetworkGetInterfaceAndNextHopFromForwardingTable(Node *node, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress);	Forwarding Table から宛先ネットワークアドレスが、desinationAddress（にマスクをかけたもの）に等しいエントリを探し、その NextHop 及び送出インタフェース番号を返す。
2	void NetworkGetInterfaceAndNextHopFromForwardingTable(Node *node,	#1 と同じだが、送出インタフェースを限定して Forwarding Table を検

	int currentInterface, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress);	索し、NextHop アドレスを取得する。
3	void NetworkGetInterfaceAndNextHopFromForwardingTable(Node *node, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress, BOOL *routeType);	#1 に加えて、Forwarding Table が特定 IP アドレス宛のエントリを持っているのか、ネットワークアドレス宛のエントリを持っているのかの情報も返す。
4	void NetworkGetInterfaceAndNextHopFromForwardingTable(Node *node, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress, BOOL testType, NetworkRoutingProtocolType type);	Routing Protocol の type を限定 又は除外して Forwarding Table を検索し、NextHop アドレスおよび送出先インタフェース番号を取得する。
5	void NetworkGetInterfaceAndNextHopFromForwardingTable(Node *node, int operatingInterface, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress, BOOL testType, NetworkRoutingProtocolType type);	#4 と同じだが、送出インタフェースを限定して Forwarding Table を検索し、NextHop アドレスを取得する。
6	void NetworkGetInterfaceAndNextHopFromForwardingTable (Node *node, NodeAddress destinationAddress, int *interfaceIndex, NodeAddress *nextHopAddress, BOOL testType, NetworkRoutingProtocolType type, BOOL* routeType)	#4 に加えて、Forwarding Table が特定 IP アドレス宛のエントリを持っているのか、ネットワークアドレス宛のエントリを持っているのかの情報も返す。

ここでは、最も基本のパターンである #1 の関数のみ説明する。

```
10276 void NetworkGetInterfaceAndNextHopFromForwardingTable(
10277     Node *node,
10278     NodeAddress destinationAddress,
10279     int *interfaceIndex,
10280     NodeAddress *nextHopAddress)
10281 {
10282     NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
10283     NetworkForwardingTable *forwardTable = &(ip->forwardTable);
10284     int i;
10285
10286     *interfaceIndex = NETWORK_UNREACHABLE;
10287     *nextHopAddress = (unsigned) NETWORK_UNREACHABLE;
10288
10289     //NetworkPrintForwardingTable(node);
10290
10291     for (i=0; i < forwardTable->size; i++) {
10292         NodeAddress maskedDestinationAddress =
10293             MaskIpAddress(
10294                 destinationAddress, forwardTable->row[i].destAddressMask);
10295
10296         if (forwardTable->row[i].destAddress == maskedDestinationAddress
10297             && forwardTable->row[i].nextHopAddress !=
10298             (unsigned) NETWORK_UNREACHABLE
10299             && forwardTable->row[i].interfaceIsEnabled != FALSE)
```

```
10300          {
10301              *interfaceIndex = forwardTable->row[i].interfaceIndex;
10302              *nextHopAddress = forwardTable->row[i].nextHopAddress;
10303              break;
10304          }
10305      }
10306 }
```

送出先インタフェース番号の取得

行番号	処理内容
10286-10287	この後の探索処理で宛先ネットワークが一致するエントリが見つからなかった場合に、呼び出し元に返すためのデフォルト値 NETWORK_UNREACHABLE を代入している。NETWORK_UNREACHABLE は network_ip.h にて -2 で#define されており、QualNet 内ではネットワーク到達不能を表すシンボルとして随所で使われている。
10291	10291 行目の for ブロックは、からフォワーディングテーブルを先頭から順番に探索を行いネットワークアドレスが一致する最初のエントリを探し NextHop を返す処理を行っている(ただし無効なインタフェースは検索対象から除外)。なお、同一宛先ネットワーク宛ての複数のエントリが存在する場合もあるが、1.3.1 で説明したように Administrative Distance(AD)値が小さい方が常に先になるようにソートされて格納されているため、自動的に AD 値の小さいエントリ(すなわち優先度の高いルーティングプロコトルが作成したエントリ)の Next-Hop が返される。
10292-10294	引数で指定された宛先 IP アドレにとエントリのサブネットマスクをかけて、ネットワークアドレスを生成している。この NW アドレスと、Forwarding Table エントリの宛先 NW アドレスが等しければ一致したとみなされる。

なお、同一サブネットや P2P Link で接続された隣接ノードへの経路は、初期化時に自動的に Forwarding Table 追加されている点に注意しよう。そのため、ルーティングプロトコルや、スタティックルートを設定しなくても、隣接のノードへはパケットが常に到達可能な状態になっている。なお、そのような隣接ノードへの経路情報については、Forwarding Table のエントリの NextHop アドレスが 0 に設定されている。

4.5 経路制御

本節では経路表の作成・更新を行う経路制御処理について、QualNet の実装コードを追いながら詳細に説明する。まず最初に、スタティック経路を設定で与える場合について解説し、その後、経路制御プロトコルを用いた動的経路制御について、プロアクティブ型経路制御とリアクティブ型経路制御の順に解説する。

4.5.1 Static Route

AODV や OSPF などの動的なルーティングプロトコル以外に、あらかじめ設定ファイルで静的な経路表を与え、それに従ったルーティングを行わせることができる。QualNet ではこの経路を「Static Route」と呼んでいる。

4.5.1.1 Static Route の設定ファイル

Static Route の設定ファイルの書き方を例を交えて説明する。

図 4-9 に各ノードの経路表の例を記す。各ノードには IP パケットをどのアドレス宛に送信すればよいかを経路表に登録されている必要がある。

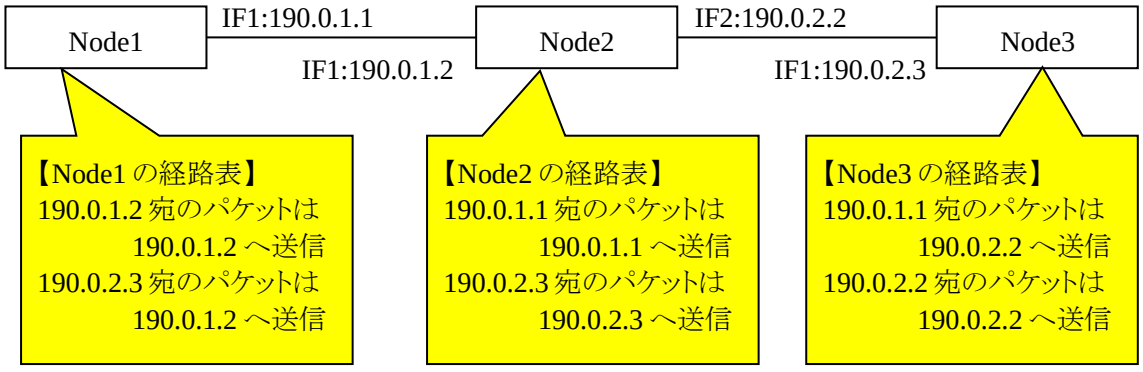

図 4-9 各ノードの経路表

上述の設定を QualNet の Static Route の設定ファイル(***.routes-static)に記述することで設定可能である。以下が上述の設定を反映させた際の記述である。

```
#NodeID       宛先アドレス    NextHop アドレス  送出 IF 番号(省略化)  コスト値(省略化)
# Node1 の設定
1             190.0.1.2     190.0.1.2
1             190.0.2.3     190.0.1.2
# Node2 の設定
2             190.0.1.1     190.0.1.1        1
2             190.0.2.3     190.0.2.3        2
# Node3 の設定
3             190.0.1.1     190.0.2.2        1
3             190.0.2.2     190.0.2.2        1                  2
```

経路を設定するノード番号に続いて、経路表の内容として宛先アドレスと NextHop のアドレスを指定する。上記の例は、IPv4 の例であるが、IPv6 も同様に指定することができる。また IPv4 を用いる場合のみ、送出 IF 番号及び当該経路のコスト値(整数)も指定することができる(省略可)。

表 4-5 Static Route 設定ファイルの設定項目

項目	説明
Node ID	経路表を設定するノード ID (整数のみ)
宛先アドレス	ホストアドレスまたはネットワークアドレス(例:N8-190.0.1)等で指定する。ノード ID は指定不可。
NextHop アドレス	NextHop の IP アドレス。ホストアドレスのみ指定可(ネットワークアドレス不可)。
送出 IF 番号 (省略化)	パケットの送出先 IF 番号が指定可能。省略時は NextHop アドレスから IF 番号が決められる。IPv6 の場合は指定不可。
コスト値 (省略化)	この経路のコスト値。省略時は 1 が割当てられる。コスト値は一部の経路制御プロトコルにより使用される。IPv6 の場合は指定不可。

4.5.1.2 QualNet の ForwardingTable への読み込み

TBD

【コラム：Static Route と Default Route と Default Gateway】

Static Route に似た機能として Default Route という機能がある。Default Route の場合も Static Route と同様に外部ファイルであらかじめ経路情報を記述し設定を行う。これらの異なる点は何かというと、経路の優先度である。

Static Route の設定が動的経路制御プロトコルにより作成されたどの経路よりも優先されるのに対し、Default Route は優先度が最も低い。優先度は前述の AD 値により表され、Static Route なら AD 値 1, Default Route なら AD 値 255 である。

ここで、Static Route、Default Route はともに、経路表のエントリとして追加されるものである点に注意しよう。Default Route 含め、経路表からパケットの Next Hop が決められなかったとき、最終的にパケットの送信先に用いられるのが Default Gateway である。

4.5.2 プロアクティブ型経路制御の例（Bellman-Ford）

Bellman-Ford(ROUTING-PROTOCOL　BELLMANFORD)は、QualNet におけるデフォルトで設定される経路制御プロトコルである。同類の経路制御プロトコルとして、RIP(ROUTING-PROTOCOL　RIP)、RIPng(ROUTING-PROTOCOL　RIPng)がある。

Bellman-Ford は、別名をフォードフルカーソンといい情報プロトコル（RIP）ルーティングの基礎となるメカニズムである。ただし、RIPv2 には準拠していない。制御パケット伝送のために UDP（User Datagram Protocol）を使用し、距離ベクトル型経路制御アルゴリズムに分類される。ほかの経路制御プロトコルに比べ非常にシンプルであるが、スケーラビリティが悪く、また経路のループが発生してしまう等の欠点がある。複数のサブネットで構成されている比較的大きなネットワークの場合は RIP を使用するのが一般的である。なお、Bellman-Ford アルゴリズムの QualNet における設定の詳細については、Developer Model Library のマニュアル 4.1 章に、RIP の詳細については、同 4.3 章に記述がある。

Bellman-Ford における NetworkUpdateForwardingTable( )関数の呼び出し箇所は、HandleCheckRouteTimeoutAlarm( )関数と、ProcessRouteAdvertisementPacket( )関数の 2 か所である。Bellman-Ford も通常の QualNet の流儀と同じで、初期化の RoutingBellmanfordInit( )関数、終了処理の RoutingBellmanfordFinalize( )関数、実際のイベント処理の RoutingBellmanfordLayer( )関数が主要な 3 関数になる。NetworkUpdateForwardingTable( )関数の呼び出しは全て RoutingBellmanfordLayer( )関数からの処理となる。以下に、呼び出し関係を示す。

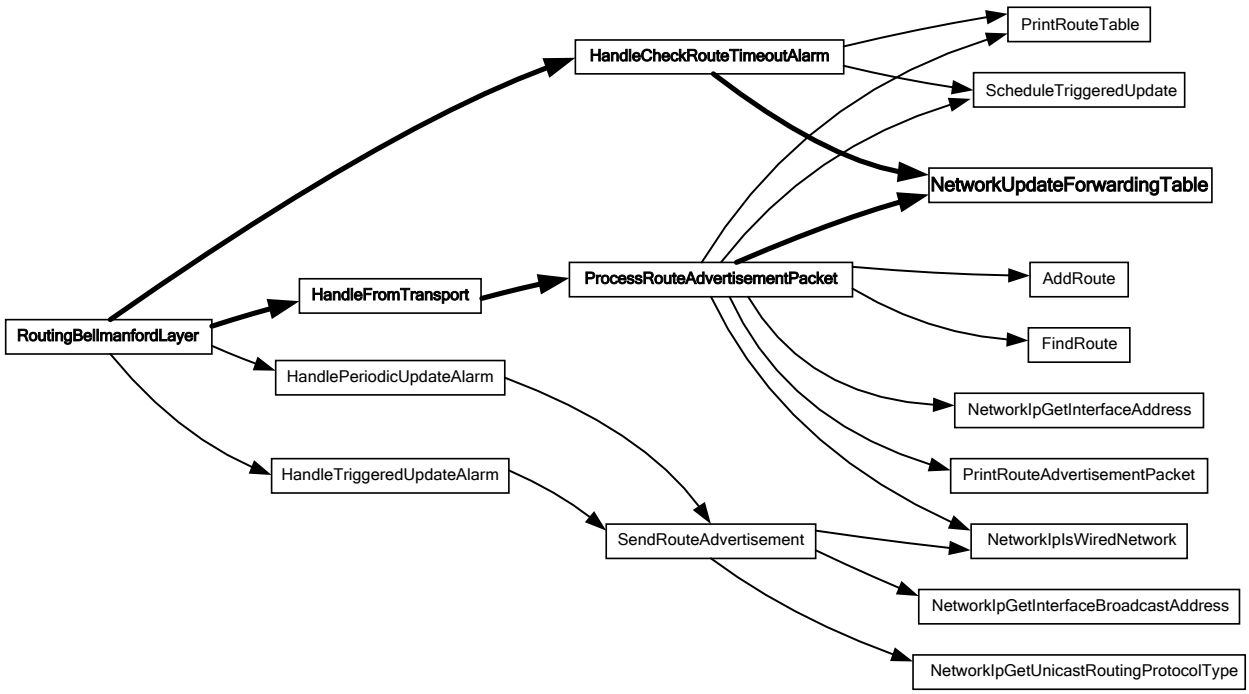

図 4-10 Bellman-Ford 関連の関数の呼び出し関係

まず、HandleCheckRouteTimeoutAlarm( )関数の内容を説明する。

routing_bellmanford.cpp

```
 754 //-----------------------------------------------------------------------
-
 755 // FUNCTION     HandleCheckRouteTimeoutAlarm
 756 // PURPOSE      Checks for route timeouts, schedules triggered update if
 757 //              necessary.  Schedules next route timeout check.
 758 //
```

```
 759 // Parameters:
 760 //
 761 // Node *node
 762 //     For current node.
 763 //
 764 // Notes:
 765 //
 766 // All routes (that aren't timed out already) are checked for staleness.
 767 // Stale routes are marked unreachable and a triggered update is
 768 // scheduled.
 769 //--------------------------------------------------------------------------
-
 770
 771 static void //inline//
 772 HandleCheckRouteTimeoutAlarm(Node *node)
 773 {
 774     Bellmanford *bellmanford = (Bellmanford *) node->appData.bellmanford;

.... 中略 ....

 841             // Update forwarding table to indicate no route to
 842             // destAddress.
 843
 844             NetworkUpdateForwardingTable(
 845                 node,
 846                 bellmanford->routeTable[i].destAddress,
 847                 bellmanford->routeTable[i].subnetMask,
 848                 (unsigned) NETWORK_UNREACHABLE,
 849                 ANY_INTERFACE,
 850                 bellmanford->routeTable[i].distance,
 851                 ROUTING_PROTOCOL_BELLMANFORD);
```

この HandleCheckRouteTimeoutAlarm( )関数は、RoutingBellmanfordLayer( )関数が受け取った Message が MSG_APP_CheckRouteTimeoutAlarm の場合に呼び出される。

次に、この MSG_APP_CheckRouteTimeoutAlarm の Message を MESSGE_Send( )している個所を調べると、初期化の RoutingBellmanfordInit( )関数と、HandleCheckRouteTimeoutAlarm( )関数(つまり自分自身)で行っている事がわかる。

関数名 HandleCheckRouteTimeoutAlarm から想像できるように、この処理はタイマ処理になっており、定期的に NetworkUpdateForwardingTable( )を更新している。

ProcessRouteAdvertisementPacket( )関数の中身は以下のようになっている。

routing_bellmanford.cpp

```
1226 //--------------------------------------------------------------------------
-
1227 // FUNCTION     ProcessRouteAdvertisementPacket
1228 // PURPOSE      Processes a route broadcast packet from UDP.
1229 //
1230 // Parameters:
1231 //
1232 // Node *node
1233 //     For current node.
1234 // NodeAddress neighborAddress
1235 //     IP address of broadcasting node.
1236 // int numAdvertisedRoutes
1237 //     Number of rows in broadcasted route table.
1238 // AdvertisedRoute *neighborRowPtr
1239 //     Pointer to broadcasted route table.
1240 //
1241 // Notes:
1242 //
1243 // Each row in the broadcasted route table is looked at, and the
```

```
1244 // local route table is updated appropriately.  The IP forwarding
1245 // table is also updated, and a triggered update may be scheduled.
1246 //-------------------------------------------------------------------------
-
1247
1248 static void //inline//
1249 ProcessRouteAdvertisementPacket(
1250     Node *node,
1251     NodeAddress neighborAddress,
1252     int incomingInterfaceIndex,
1253     int numAdvertisedRoutes,
1254     AdvertisedRoute *neighborRowPtr)
1255 {
1256     Bellmanford *bellmanford = (Bellmanford *) node->appData.bellmanford;

.... 中略 ....

1442             // Update forwarding table.
1443
1444             NetworkUpdateForwardingTable(
1445                 node,
1446                 destAddress,
1447                 subnetMask,
1448                 rowPtr->nextHop,
1449                 rowPtr->outgoingInterface,
1450                 rowPtr->distance,
1451                 ROUTING_PROTOCOL_BELLMANFORD);
```

この ProcessRouteAdvertisementPacket( )関数は、HandleFromTransport( )関数からのみ呼び出される。そして、HandleFromTransport( )関数は、RoutingBellmanfordLayer( )関数が受け取った Message が MSG_APP_FromTransport の場合に呼び出される。

MSG_APP_FromTransport の Messase は Bellman-Ford 専用ではない。以下の api.h での定義を見ればわかるように、UDP パケットの処理（TransportUdpSendToApp( )関数）で使われる。

api.h

```
 273     MSG_APP_FromTransCloseResult                = 604,
 274     MSG_APP_TimerExpired                        = 605,
 275     MSG_APP_SessionStatus                       = 606,
 276     /* Messages Types for Application layer from UDP */
 277     MSG_APP_FromTransport                       = 610,
 278
 279     /* Messages Types for Application layer from NS TCP */
 280     MSG_APP_NextPkt                             = 620,
 281     MSG_APP_SetupConnection                     = 621,
```

Bellman-Ford では情報交換に UDP パケットを使用するので、ProcessRouteAdvertisementPacket( )関数が受け取る情報は、他の Node からの Bellman-Ford 情報であることがわかる。

まとめると、HandleCheckRouteTimeoutAlarm( )関数で更新する NetworkUpdateForwardingTable( )は、自らの情報を定期的に更新する処理である。
一方、ProcessRouteAdvertisementPacket( )関数で更新する NetworkUpdateForwardingTable( )は、他の Node からの情報により自らの NetworkUpdateForwardingTable( )を更新する処理である。

4.5.3 リアクティブ型経路制御の例（AODV）

TBD