QualNet
Building Smarter Networks

QualNet Hacks

＃２イベント処理

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 を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

2 イベント処理

2.1 離散シミュレーションとは

QualNet は離散事象型シミュレータである。離散事象型シミュレーションとは、待ち行列型モデルの混雑現象を分析・評価するために、様々な現象に適用されている。

事象(イベント)とは、システムに状態変化を起こす瞬間的な出来事を指し、例えればパケットの到達やルーティングプロトコルにおける近隣ノードへの経路変更の通知などが挙げられる。また、イベントが発生することで行われる処理の例としては、隣接レイヤへのパケットの送信、状態変数の更新、タイマの開始や再開などが挙げられる。

概念的に説明すると、
イベント a,b,c がその発生時刻順にスケジュールされているとする。
まず、発生予定時刻の最も早いイベント a が、ある時間(t0)に処理され、

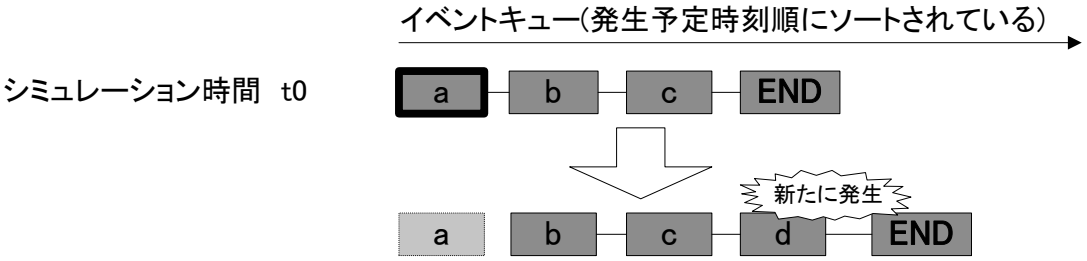

図 1 離散事象型シミュレーション概念図(状態 1)

その結果を受けて、イベント c より後の発生予定時刻の、ある新たなイベント d が生成されたとする。
これにより、イベントキューが変更され、イベント c の後にイベント d がスケジュールされる。

次に発生が予定されているイベント b のシミュレーション系内時刻が t0+Δt であるとすると、QualNet は実時間と関係なく、シミュレーション系内時刻を Δt だけ進めて、イベント b を処理する。

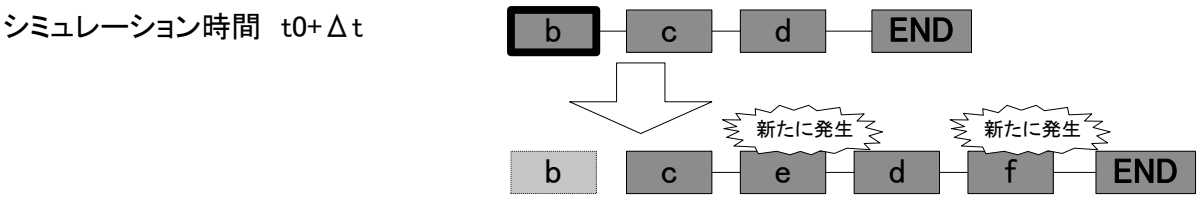

図 2 離散事象型シミュレーション概念図(状態 2)

その結果を受けて、イベント c より後の発生予定時刻である新たなイベント e と、イベント d より後の発生予定時刻であるイベント f が生成されたとする。つまり、イベントキューが上図のように更新されることになる。

このように、イベントキューの中で発生予定時刻が最早のイベントを取り出し、その処理によるイベントキューの更新とシステムの状態変化を延々と繰り返し、キューの中のイベントがなくなるまで処理していく。
また、上述の通り、シミュレーション系内時間を保持し、時間的に隣りあうイベント間ではシステムの状態が変化しないと考え、シミュレーション系内時間を進めて次の発生イベントを処理するので、時間の進み方は不均等となる。

2.2 イベントスケジューラ

QualNet は、図 3 に示すように、ノードインスタンスなどが配置されるシミュレーションフィールドと、そのバックエンドでイベントを処理するイベントスケジューラで構成される。ただし、QualNet をマルチスレッド実行（並列処理オプションを用いて実行）する際には、シミュレーションフィールドは Partition と呼ばれるフィールド分割単位で区切られ、またイベントスケジューラはその Partition 単位で用意されるので、より正確に言えば、QualNet は複数の「Partition+イベントスケジューラ」の組み合わせで構成されると言える。
なおこのイベントスケジューラは「QualNet カーネル」とも呼ばれており、まさに QualNet の特長である高速・高精度シミュレーションを実現するための工夫が凝らされた QualNet の核をなすモジュールである。

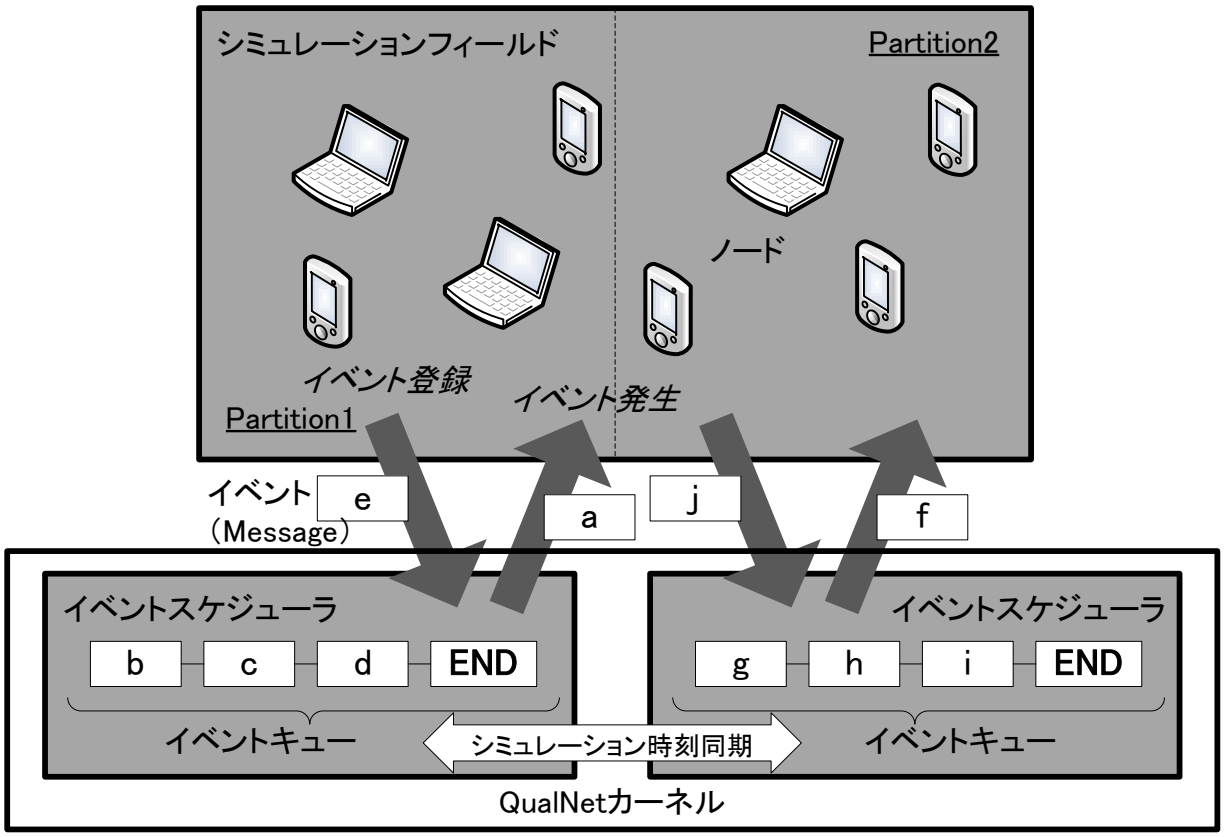

図 3　シミュレーションフィールドとイベントスケジューラ

しかし残念ながら、このイベントスケジューラのソースコードは公開されておらず、バイナリ形態でのみ提供されている。ただし、イベントキューの構造など一部は表 1 に示すようにソースコード形態でも提供されており、ここではこれらの情報から垣間見える QualNet のイベントスケジューラの概観について解説する。

表 1 イベントスケジューラ関連ソースコード

ファイル名称	説明
include/scheduler_types.h	イベントキューの種別とデータ構造を定義。
include/scheduler.h	イベントスケジューラ API 定義。
include/calendar.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。
include/simplesplay.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。
include/splaytree.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。
include/sched_calendar.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。
include/sched_splaytree.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。
include/sched_std_library.h	旧バージョンのスケジューラ実装コードの一部と見受けられる。 Ver.5.1 では使用されていないようなので説明しない。

まず始めに注意点として挙げておくと、表 1 を見て分かるようにイベントスケジューラは、実装上 scheduler という名前で呼ばれている。一方で QualNet にはプロトコルスタック内で用いるキューとそのキューを操作するスケジューラに関するソースコードが提供されており、実装上は if_queue や if_scheduler という名前で呼ばれている。”if_”という接頭辞が付いているのは、ネットワークインタフェースの送受信キューとして用いることを想定しているためである。一般にプロトコル開発を進める際には、本節で説明するイベントスケジューラについて意識する必要はないため、通常、プロトコル開発者が「スケジューラ」と表現する際には本章で取り扱う scheduler ではなく if_scheduler のほうを指すことになるので注意する必要がある。

では本題に入るが、イベントスケジューラ本体を表すデータ構造は、SchedulerInfo という構造体で定義されている。この構造体は、イベントキュー種別を表す schedQueueType メンバ変数、イベントキュー実装の 1 形態であるスプレー木構造へのアクセスポインタである splayTree メンバ変数、同じくイベントキュー実装の 1 形態であるカレンダーキュー構造へのアクセスポインタである calendarQ メンバ変数、おそらく統計情報収集可否を表すフラグと推測される isCollectingStats メンバ変数、その他イベントキューの各種操作関数へアクセスするための関数ポインタ群で構成される。

なお、イベントスケジューラの実体(インスタンス)へは、Partition インスタンスあるいは Node インスタンス内にある schedulerInfo メンバ変数からアクセスできるようになっている。

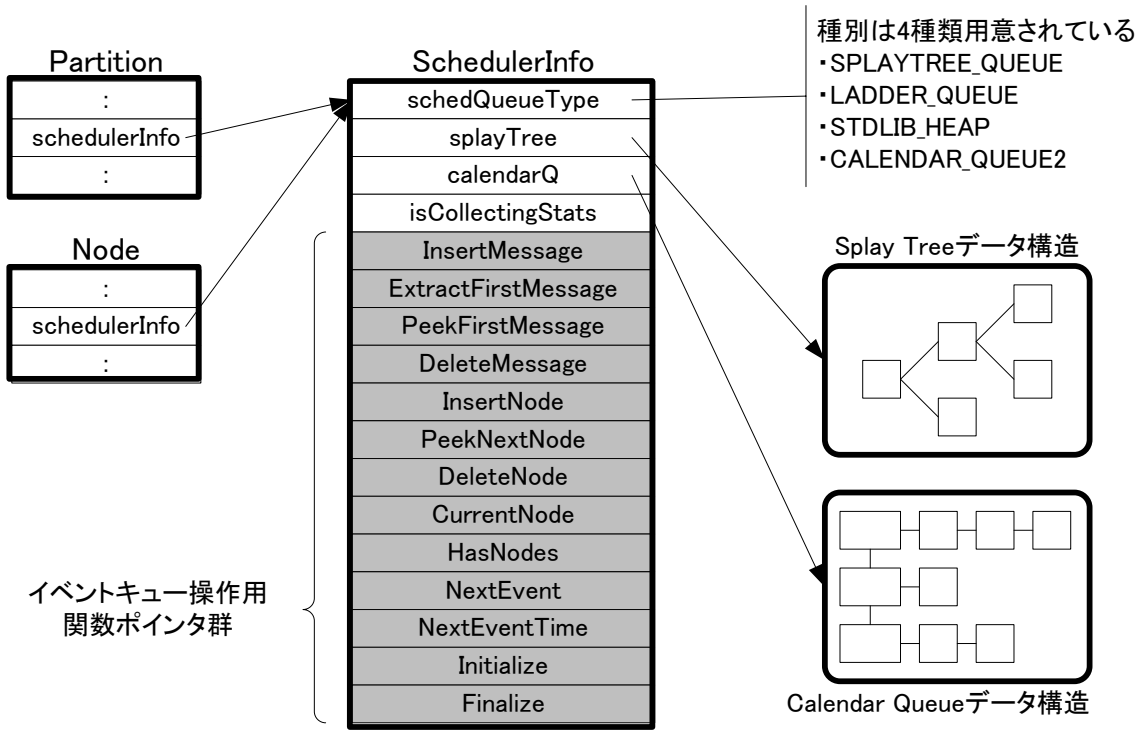

図 4 イベントスケジューラのデータ構造

イベントキュータイプは、以下のように SchedulerQueueType 列挙型で定義されており、4 種類の実装方式があることが確認できる。

scheduler_types.h

```
170 // -------------------- Queue Types available ---------------------
171 typedef enum {
172     SPLAYTREE_QUEUE,
173     // CALENDAR_QUEUE,      // NO LONGER USED
174     LADDER_QUEUE,
175     STDLIB_HEAP,
176     CALENDAR_QUEUE2
177 } SchedulerQueueType;
```

QualNet ソースコード中の適当な関数(例えば NODE_ProcessEvent( )関数内)で node->schedulerInfo->schedQueueType の値を printf( )関数を使って出力させてみたところ、Ver.5.1 においてはデフォルトで CALEN1DAR_QUEUE2 が使用されていることが確認できた。またその他のイベントキュー種別への変更方法は、ユーザーズガイドやプログラマーズガイド等の公式なマニュアルには掲載されていないが、scenario ディレクトリ配下の default シナリオの設定ファイル default.config を見てみると、以下のような SCHEDULER-QUEUE-TYPE という設定パラメータがあり、このパラメータで変更することが可能であることが分かる。実際に試してみたところ、CALENDAR、SPLAYTREE、STDLIB という値で設定できた。(残念ながら LADDER については認識されなかった。)ただし、普通に QualNet を使うにあたっては、このパラメータをデフォルト値から変更する必要が生じることはまずなく変更する意義も見いだせないので、以上の話はあくまで知識として留めておく程度の話だと認識してもらってよい。

なお、この default.config には、QualNet の設定ファイルで設定可能な全パラメータについての解説が下の例のようにコメント形式で記載されているので一度じっくり見てみることをお勧めする。

```
5012 ##########################################################################
5013 # Scheduler                                                              #
5014 ##########################################################################
5015 # The following tells the scheduler what type of queue to use when
5016 # scheduling events.
5017 #
5018 # Use the following to change the scheduler's queue type:
5019 #
5020 # SCHEDULER-QUEUE-TYPE          SPLAYTREE | CALENDAR
5021 #
5022 # By default, the scheduler's queue type is SPLAYTREE.
5023 # Uncomment this to enable the Calendar queue
5024 #SCHEDULER-QUEUE-TYPE            CALENDAR
5025 SCHEDULER-QUEUE-TYPE            SPLAYTREE
```

最後に、QualNet のイベントスケジューラが用意しているプログラミング API の一覧を表 2 に示す。これらの API は、プロトコル実装を進める上で直接利用する機会は少ないと思われるが、後の章でも出てくる Message 操作関数などからも呼び出されており、その存在を理解しておくことは QualNet を使いこなしていく上で有用である。なお、これらの API は全て include/scheduler.h ファイルで定義されている。

表 2 イベントスケジューラ API

API 関数	説明
SCHED_InsertMessage	イベントスケジューラに新規イベントを登録する際に使用する。
SCHED_ExtractFirstMessage	ある Node に関する先頭イベントを取り出す際に使用する。 (SCHED_NextEvent と同じ効果が得られるように見受けられるが詳細は不明)
SCHED_PeekFirstMessage	ある Node に関する先頭イベントを覗く際に使用する。
SCHED_DeleteMessage	あるノードに関するあるイベントを削除する際に使用する。
SCHED_InsertNode	イベントスケジューラに新規 Node を登録する際に使用する。
SCHED_PeekNextNode	次に処理されるイベントの対象 Node を覗く際に使用する。
SCHED_DeleteNode	イベントスケジューラからある Node を削除する際に使用する。
SCHED_CurrentNode	現在処理中イベントの対象 Node を取得する際に使用する。
SCHED_HasNodes	イベントスケジューラに Node が登録されているかどうかを調べる際に使用する。
SCHED_NextEvent	ある Node に関する次のイベントを取り出す際に使用する。
SCHED_NextEventTime	次のイベント発生時刻を取り出す際に使用する。
SCHED_Initalize	イベントスケジューラ初期化 API
SCHED_Finalize	イベントスケジューラ終了処理 API

2.3 イベントの実装

QualNet におけるイベントは、Message というクラス構造を用いて実装されている。そのため、Message の作成と破棄を行う機会は非常に多い。実際、QualNet のソースコードを検索すると、Message の作成 API である MESSAGE_Alloc 関数や、破棄を行う関数 MESSAGE_Free 関数を用いている箇所が非常に多いことが確認できる。本節では、QualNet におけるイベントの実体である Message について解説する。

2.3.1 Message のライフサイクル

Message の種類については後述するが、Message には、単純なタイマに用いるためのサイズの小さい(ほぼ空)物も存在するが、通常はレイヤ間のパケットのやり取りを行うために、一定のメモリ領域を占有する。単純に考えれば、Message の作成の度に malloc 関数や operator new を用いてメモリを確保し、

Message の破棄の度に free 関数や operator delete を用いてメモリを解放することになる。しかし、メモリ確保処理は一般に負荷のかかる処理であるため、QualNet では Messsage 用のメモリプールを用意し、そこに確保された Message を使いまわす(リサイクルする)ことで、頻繁なメモリの割当と解放を避けている。以下に、Message のライフサイクルの図を示す。

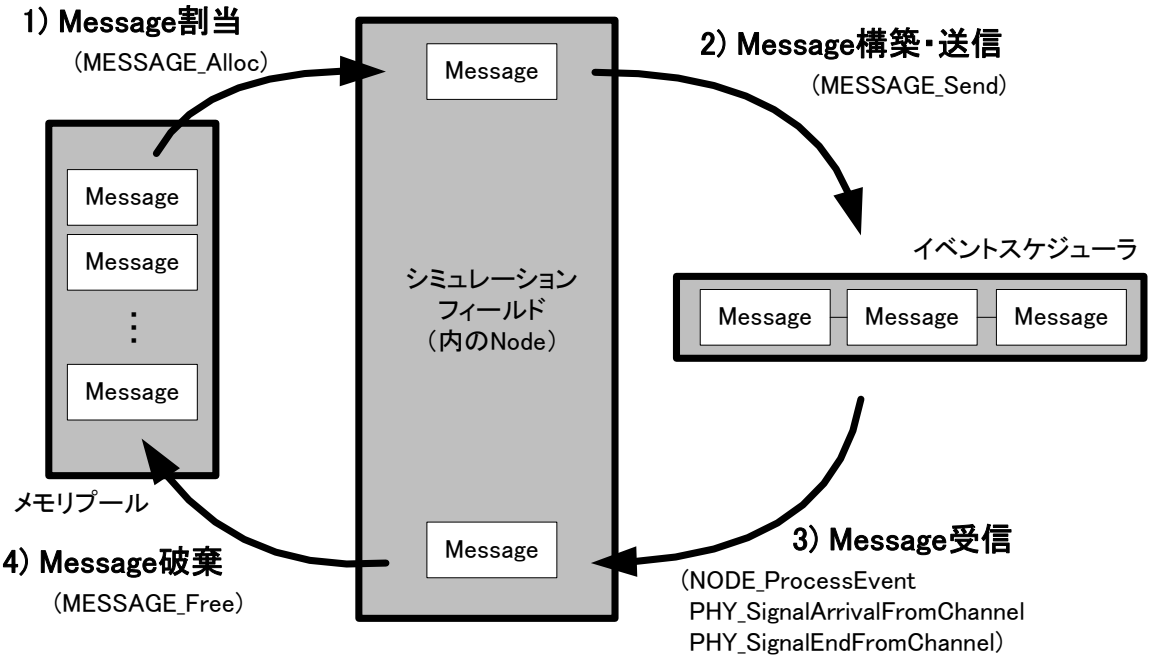

図 5　Message のライフサイクル

1) Message 割当
Message を作成するときには、MESSAGE_Alloc 関数を用いる。MESSAGE_Alloc 関数の内部では、基本的にメモリの確保は行わずにメモリプールからの Message 取得が行われる。
2) Message 構築・送信
Message にイベント付随情報を適宜詰め込み、MESSAGE_Send 関数を用いてイベントスケジューラへ Message 送信を行う(イベントの登録)。
3) Message 受信
イベントスケジューラがイベントを発生させると、そのイベントに対応する Message がシミュレーションフィールド上の対象 Node に配送される。イベントの配送はイベントスケジューラがイベントハンドリング関数を呼び出すことで行われる。イベント配送を受けた Node は、イベントスケジューラから受信した Message に対応する処理を適宜行う。イベントスケジューラが呼び出すイベントハンドリング関数としては、NODE_ProcessEvent 関数、PHY_SignalArrivalFromChannel 関数、PHY_SignalEndFromChannel 関数の 3 種類が存在し、発生するイベント種別に応じて使い分けられる。
4) Message 破棄
MESSAGE_Free 関数を用いて不要となった Message を破棄する。MESSAGE_Free 関数では、メモリ領域の解放は行わず、メモリプールへの Message の返却のみ行う。

ここで、メモリプールからの出し入れ処理と Message 送信処理をもう少し詳しく見てみることにする。

2.3.2 Message 割当処理（MESSAGE_Alloc 関数）

以下に、MESSAGE_Alloc 関数のコードを示す。536-547 行目を見て分かる通り、メモリプールの実体は、partition->msgFreeList である。ここでは、partition->msgFreeList が空の場合(つまりメモリプールに未割当 Message が存在しない場合)のみ operator new を用いて新規に Message を確保し、それ以外の場合(つまりメモリプールに未割当 Message が存在する場合)にはそのリストから Message を取り出して使用している様子が確認できる。

message.cpp

```
516 Message* MESSAGE_Alloc(PartitionData *partition,
517                        int layerType,
518                        int protocol,
519                        int eventType,
520                        bool isMT)
521 {
522     Message *newMsg = NULL;
523
524 #ifdef MEMDEBUG
525     ++allocs;
526     if (!(allocs % 100))
527     printf("partition %d, allocs = %d, deallocs = %d\n",
528            partition->partitionId, allocs, deallocs);
529 #endif
530
531 #ifdef MESSAGE_NO_RECYCLE
532     newMsg = new Message();
533 #else
534     // When called from a worker context, we can't use the partition
535     // data FreeList (it isn't locked)
536     if ((partition->msgFreeList == NULL) || (isMT))
537     {
538         newMsg = new Message();
539         newMsg->infoArray.reserve(10);
540     }
541     else
542     {
543         newMsg = partition->msgFreeList;
544         partition->msgFreeList =
545             partition->msgFreeList->next;
546         (partition->msgFreeListNum)--;
547     }
548 #endif
549     assert(newMsg != NULL);
550
551     newMsg->initialize(partition);
552
553     // Set layer, protocol, event type
554     MESSAGE_SetLayer(newMsg, (short) layerType, (short) protocol);
555     MESSAGE_SetEvent(newMsg, (short) eventType);
556
557     // Set multi threaded
558     if (isMT)
559     {
560         newMsg->mtWasMT = TRUE;
561     }
562
563     return newMsg;
564 }
```

2.3.3 Message 破棄処理（MESSAGE_Free 関数）

以下に、MESSAGE_Free 関数のコードを示す。1592-1593 行目で partition->msgFreeList の先頭に今から不要となる Message へのポインタを代入していることが分かる。ここがメモリプールへ Message を戻す処理を行っている部分である。なお 1586 行目の条件式からもわかるように、メモリプールに保持可能な未割当 Message の数は、MSG_LIST_MAX 個(デフォルトでは 10000)である。

message.cpp

```
1559 void MESSAGE_Free(PartitionData *partition, Message *msg)
1560 {
1561 #ifdef MESSAGE_NO_RECYCLE
1562     // Make sure we always go through the delete operator when not
1563     // recycling messages (delete does not recycle messages)
1564     if (!msg->getDeleted())
1565     {
1566         delete msg;
1567         return;
1568     }
1569 #endif
1570
1571 #ifdef MEMDEBUG
1572     deallocs++;
1573     //printf("partition %d, allocs = %d, deallocs = %d\n", partition-
>partitionId, allocs, deallocs);
1574 #endif
1575
1576     bool wasMT = msg->mtWasMT;
1577
1578     ERROR_Assert(!msg->getFreed(), "Message already freed");
1579     ERROR_Assert(!msg->getDeleted(), "Message already deleted");
1580     ERROR_Assert(!msg->getSent(), "Freeing a sent message");
1581
1582 #ifndef MESSAGE_NO_RECYCLE
1583     // Message recycling is enabled
1584     if ((partition != NULL) &&
1585         (wasMT == false) &&
1586         (partition->msgFreeListNum < MSG_LIST_MAX))
1587     {
1588         // Free contents and save to list
1589         MESSAGE_FreeContents(partition, msg);
1590         msg->setFreed(true);
1591
1592         msg->next = partition->msgFreeList;
1593         partition->msgFreeList = msg;
1594         (partition->msgFreeListNum)++;
1595     }
1596     else
1597     {
1598         // Delete message completely
1599         delete msg;
1600     }
1601 #endif
1602 }
```

ところで、メモリプールに関連する処理が#ifndef MESSAGE_NO_RECYCLE で囲まれていることからも推察できる通り、実は QualNet のコンパイル時にメモリのリサイクルを行わないように設定することも可能である。

message.cpp

```
  ... 略 ...
52 #define DEBUG 0
53 #define noMESSAGE_NO_RECYCLE
54 //#define MESSAGE_NO_RECYCLE
55 /* Maximum amount of messages that can be kept in the free list. */
```

```
56 #define MSG_LIST_MAX 10000
57 //#define MSG_LIST_MAX 0
   ... 略 ...
```

54 行目の#define MESSAGE_NO_RECYCLE のコメントを外すことで、メッセージのリサイクルを行わないようにすることができる。(もちろん、再コンパイルは必要)。

Message のリサイクルが有効である場合、ある場所での Message への不正な操作の影響が、まったく関連のない場所に発現することがあり、この種のバグの原因究明は非常にやっかいな作業となる。まずは、プログラミング・デバッグを行う際は、Message はリサイクルされるものだということを知っておくべきである。また、時として、メッセージのリサイクルを行わない設定が、不正なメモリアクセス等のバグ検出に役立つことがある。メモリ不正アクセス系のバグの可能性があると思われるときには、試してみることをお勧めする。

2.3.4 Message 送信処理（MESSAGE_Send 関数）

以下に、MESSAGE_Send 関数のコードを示す。2475 行目まではエラーチェック処理である。2477-2510 行目は QualNet をマルチスレッド実行した際の処理で、この場合は登録しようとしている Message を当該ノードが属する partition の sendMTList というリストに挿入している。このリストに挿入された Message は、後刻 partition 内のノードに関する一括処理をする際に SCHED_InsertMessage 関数を用いてイベントスケジューラに登録されるがここではその説明は省略する(詳細については、partition.cpp ファイルの PARTITION_ProcessSendMT 関数を参照)。2511-2514 行目は QualNet をシングルスレッド実行した際の処理で、この場合は SCHED_InsertMessage 関数を用いてイベントスケジューラに Message を登録している。

message.cpp

```
2423 void MESSAGE_Send(Node *node, Message *msg, clocktype delay, bool isMT) {
2424     if (DEBUG)
2425     {
2426         printf("at %" TYPES_64BITFMT "d: partition %d node %3d scheduling
event %3d at "
2427                "time %" TYPES_64BITFMT "d on interface %2d\n",
2428                getSimTime(node),
2429                node->partitionData->partitionId,
2430                node->nodeId,
2431                msg->eventType,
2432                getSimTime(node) + delay,
2433                msg->instanceId);
2434         fflush(stdout);
2435     }
2436
2437     if (delay < 0)
2438     {
2439         char errorStr[MAX_STRING_LENGTH];
2440         char delayStr[MAX_STRING_LENGTH];
2441         TIME_PrintClockInSecond (delay, delayStr);
2442         sprintf(errorStr,
2443             "Node %d sending message with invalid negative delay of %s\n",
2444             node->nodeId,
2445             delayStr);
2446         ERROR_ReportError(errorStr);
2447     }
2448     else if (delay == CLOCKTYPE_MAX)
2449     {
2450         char errorStr[MAX_STRING_LENGTH];
2451         sprintf(errorStr,
2452             "Node %d sending message with invalid delay of CLOCKTYPE_MAX\n",
2453             node->nodeId);
2454         ERROR_ReportError(errorStr);
```

```
2455     }
2456
2457 #ifdef USE_MPI //Parallel
2458     if (node->partitionId != node->partitionData->partitionId) {
2459         // don't schedule this because it's a remote node.
2460         MESSAGE_Free(node, msg);
2461         return;
2462     }
2463 #endif //endParallel
2464
2465 #ifdef DEBUG_EVENT_TRACE
2466     printf("at %lld, p%d, node %d scheduling %d on intf %d, delay is %lld\n",
2467            getSimTime(node), node->partitionData->partitionId, node->nodeId,
2468            msg->eventType, msg->instanceId, delay);
2469 #endif
2470
2471     ERROR_Assert(!msg->getFreed(), "Sending a freed message");
2472     ERROR_Assert(!msg->getDeleted(), "Sending a deleted message");
2473     ERROR_Assert(!msg->getSent(), "Sending an already scheduled message");
2474     msg->setSent(true);
2475
2476     msg->naturalOrder = node->partitionData->eventSequence++;
2477     if (isMT)
2478     {
2479         msg->nodeId = node->nodeId;
2480         // calculate the requested event time
2481         msg->eventTime = node->partitionData->theCurrentTime + delay;
2482 #ifdef USE_LOCK_FREE_QUEUE
2483         node->partitionData->externalLFMsgQ->pushBack(msg);
2484 #else
2485         {
2486             // Rich, testing externalLFMsgQ
2487             if (DEBUG)
2488             {
2489                 printf("at %" TYPES_64BITFMT "d: partition %d node %3d
sending"
2490                        " thread event %3d with delay %" TYPES_64BITFMT "d on"
2491                        " interface %2d\n",
2492                        getSimTime(node),
2493                        node->partitionData->partitionId,
2494                        node->nodeId,
2495                        msg->eventType,
2496                        delay,
2497                        msg->instanceId);
2498                 fflush(stdout);
2499             }
2500             // lock the list of pending messages, hopefully the lock
disappears
2501             // outside this little block.
2502             QNThreadLock messageListLock(node->partitionData-
>sendMTListMutex);
2503             // add
2504             node->partitionData->sendMTList->push_back(msg);
2505         }
2506         // list is now unlocked
2507         // Now, partition private (or some other code that is executing as
2508         // part of simulation thread needs to call PARTTIONT_SendMTPrcoess ()
2509 #endif // USE_LOCK_FREE_QUEUE
2510     }
2511     else
2512     {
2513         SCHED_InsertMessage(node, msg, delay);
2514     }
2515
2516
2517     MESSAGE_DebugSend(node->partitionData, node, msg);
2518 }
```

2.3.5 Message 操作 API 一覧

表 3 に、Message を操作するために QualNet で用意されている API 関数の一覧を示す。これらの API のうちのいくつかは既にその実装コードを説明したが、その他の API についての詳しい説明は省略する。

表 3 Message 処理 API

API 関数名称	説明
MESSAGE_AddHeader	Header を追加。
MESSAGE_AddInfo	Info を追加。
MESSAGE_AddVirtualPayload	virtualPayloadSize を指定サイズ増やす。
MESSAGE_Alloc	新規に領域を確保。
MESSAGE_AllocMT	マルチスレッドセーフな MESSAGE_Alloc。
MESSAGE_AllowLooseScheduling	return (msg->allowLoose);
MESSAGE_AppendInfo	Info を追加。
MESSAGE_CancelSelfMsg	キャンセル。
MESSAGE_CopyInfo	Info をコピー。
MESSAGE_Duplicate	複製。
MESSAGE_DuplicateMT	マルチスレッドセーフな MESSAGE_Duplicate。
MESSAGE_ExpandPacket	Packet 領域を拡張。
MESSAGE_FragmentPacket	Packet をフラグメント。
MESSAGE_Free	領域を解放。
MESSAGE_FreeContents	Contens を解放。
MESSAGE_FreeList	リスト状になった複数 Message をまとめて解放する。
MESSAGE_FreeMT	マルチスレッドセーフな MESSAGE_Free。
MESSAGE_GetEvent	当該 Message に設定されたイベント種別を返す。
MESSAGE_GetInstanceId	当該 Message に設定されたインスタンス番号を返す。
MESSAGE_GetLayer	当該 Message に設定された所属レイヤ種別を返す。
MESSAGE_GetPacketCreationTime	当該 Message に設定されたパケット生成時刻を返す。
MESSAGE_GetProtocol	当該 Message に設定された所属プロトコル種別を返す。
MESSAGE_InfoAlloc	Info 領域を確保。
MESSAGE_InfoFieldAlloc	InfoField 領域の確保。
MESSAGE_InfoFieldAllocMT	マルチスレッドセーフな MESSAGE_InfoFieldAlloc。
MESSAGE_InfoFieldFree	InfoField 領域の解放。
MESSAGE_InfoFieldFreeMT	マルチスレッドセーフな MESSAGE_InfoFieldFree。
MESSAGE_PackMessage	複数の Message を 1 つにパッキングする。
MESSAGE_PacketAlloc	Packet 領域を確保。
MESSAGE_PayloadAlloc	Payload 領域を確保。
MESSAGE_PayloadFree	Payload 領域を解放。
MESSAGE_PayloadFreeMT	マルチスレッドセーフな MESSAGE_PayloadFree。
MESSAGE_PrintMessage	デバッグ用の出力。
MESSAGE_ReassemblePacket	フラグメントされた Message の再構成。
MESSAGE_RemoteSend	異なるスレッドの Node に送信。
MESSAGE_RemoteSendSafely	異なるスレッドの Node に安全に送信。
MESSAGE_RemoveHeader	Hrader を外す。
MESSAGE_RemoveInfo	Info を外す。
MESSAGE_RemoveVirtualPayload	virtualPayloadSize を指定サイズ減らす。
MESSAGE_ReturnFragNumInfos	フラグメント化された Info 数。
MESSAGE_ReturnFragSeqNum	フラグメント化された Info のシーケンス番号。
MESSAGE_ReturnFragSize	フラグメントサイズ。
MESSAGE_ReturnHeader	Header 領域のポインタ。
MESSAGE_ReturnInfo	Info 領域のポインタ。
MESSAGE_ReturnInfoSize	Info 領域のサイズ。
MESSAGE_ReturnNumFrags	フラグメント数。
MESSAGE_ReturnPacket	Packet 領域のポインタを返す。
MESSAGE_ReturnPacketSize	packetSize を返す。
MESSAGE_RouteReceivedRemoteEvent	他スレッドから受け取る。

MESSAGE_Send	QualNet のカーネルスケジューラに登録。
MESSAGE_SendAsEarlyAsPossible	安全に急いで送信。
MESSAGE_SendMT	マルチスレッドセーフな MESSAGE_Send。
MESSAGE_Serialize	Message をバッファに詰める。
MESSAGE_SerializeMsgList	連続して Message をバッファに詰める。
MESSAGE_SetEvent	当該 Message のイベント種別を設定する。
MESSAGE_SetInstanceId	当該 Message のインスタンス番号を設定する。
MESSAGE_SetLayer	当該 Message が所属するレイヤ種別とプロトコル種別を設定する。
MESSAGE_SetLooseScheduling	msg->allowLoose = true;
MESSAGE_ShrinkPacket	パケットを短く。
MESSAGE_SizeOf	return sizeof(Message);
MESSAGE_UnpackMessage	Pack された Message を分解。
MESSAGE_Unserialize	バッファから Message を取り出す。
MESSAGE_UnserializeMsgList	連続してバッファから Message を取り出す。

2.3.6 Message 構造の詳細

以下では、QualNet で定義されている Message 構造の詳細について、実際のコードを見ながら解説する。

QualNet で使用するメッセージ種別（types）は、include/api.h で以下のように定義されている。

message.h

```
  55 // /**
  56 // ENUM        :: MESSAGE/EVENT
  57 // DESCRIPTION :: Event/message types exchanged in the simulation
  58 // **/
  59 enum
  60 {
  61     /* Special message types used for internal design. */
  62     MSG_SPECIAL_Timer                           = 0,
  63
  64     /* Message Types for Environmental Effects/Weather */
  65     MSG_WEATHER_MobilityTimerExpired            = 50,
  66
  67     /* Message Types for Channel layer */
  68     MSG_PROP_SignalArrival                      = 100,
  69     MSG_PROP_SignalEnd                          = 101,
  70     MSG_PROP_SignalReleased                     = 102,

... 略 ...

1014     //Firewall Msg
1015     MSG_Firewall_Rule,
1016
1017     /*
1018      * Any other message types which have to be added should be added before
1019      * MSG_DEFAULT. Otherwise the program will not work correctly.
1020      */
1021     MSG_DEFAULT                                 = 10000
1022 };
```

これらのメッセージは次の 3 つに分類することができる。

(ア) タイマメッセージ
(イ) レイヤ間でやり取りするパケットメッセージ
(ウ) Propagation レイヤを介してノード間でやり取りする信号メッセージ

これらのメッセージの使い方については 2.4 節で詳しく説明する。

2.3.6.1 メッセージの構成

メッセージは主に管理領域とデータ領域で構成されている。
管理領域は QualNet 内部でメッセージを管理するための情報で、直接参照することは少ない。
データ領域はメッセージの種別やパケットデータ等の情報で、必要に応じて情報を操作する。
このデータ領域は静的な領域と動的割り付け領域に分類される。
動的な領域はポインタ管理されているため、メッセージのコピーなどは専用の手続き(関数)が必要である。

【コラム 】

> QualNet 5.1 において、メッセージ本体の定義に関する大きな変更が加わった。
>
> QualNet 5.0.2 までにおける Message 定義は、「typedef struct message_str Message;」
> QualNet 5.1 における Message 定義は「class Message {};」
>
> MESSAGE API ではこの変更に対応しているため互換性があるが、QualNet5.0.2 以前のバージョンで Message を直接操作しているようなコードを QualNet 5.1 以降のバージョンに移植する場合は、注意が必要である。
> (というよりも、基本的に Message を直接操作することは避けるべきである。)

2.3.6.2 packet と payload の違い

MESSAGE_PacketAlloc 関数の実態は内部で MESSAGE_PayloadAlloc 関数を呼び出している。

message.cpp

```
1338 void MESSAGE_PacketAlloc(PartitionData *partition,
1339                           Message *msg,
1340                           int packetSize,
1341                           TraceProtocolType originatingProtocol,
1342                           bool isMT)
1343 {
1344     assert(msg->payload == NULL);
1345     assert(msg->payloadSize == 0);
1346     assert(msg->packetSize == 0);
1347     assert(msg->packet == NULL);
```

この関数内で以下の呼び出しが行われる。

message.cpp

```
1349     msg->payload = MESSAGE_PayloadAlloc(
1350                         partition,
1351                         packetSize + MSG_MAX_HDR_SIZE, isMT);
```

つまり、パケットサイズ(**packetSize**)にヘッダサイズ(**MSG_MAX_HDR_SIZE (512 で定義)**)を加えたメモリ領域が Playload の領域になる。

2.3.6.3 MESSAGE_ReturnPacketSize 関数の実体

パケットのサイズを調べる関数は以下の 3 つがある。

message.h

```
434 #define MESSAGE_ReturnPacketSize(msg) (msg->returnPacketSize())
435 #define MESSAGE_ReturnActualPacketSize(msg) (msg->returnActualSize())
436 #define MESSAGE_ReturnVirtualPacketSize(msg) (msg->returnVirtualSize())
```

実態は class Message で定義されている、次の関数である。

message.h

```
386  /// returns the packet size, including virtual data.
387  int returnPacketSize() const { return packetSize + virtualPayloadSize; }
388
389  /// returns the size the packet is supposed to represent in cases where
390  /// the implementation differs from the standard
391  int returnActualSize() const { return (isPacked)? actualPktSize : packetSize;}
392
393  /// returns the amount of virtual data in the packet
394  int returnVirtualSize() const { return virtualPayloadSize; }
```

このように PacketSize には packetSize、actualPktSize、virtualPayloadize の 3 種類があるので、Packet と Payload の違いを理解した上でサイズを調べる必要がある。

2.3.6.4 info と header の違い

info 領域は Vector で管理され、header 領域は固定サイズの配列で管理されている。

message.h

```
 45 #define MSG_MAX_HDR_SIZE                                512
 85 #define MAX_HEADERS                                     10
    ... 中略 ...
228 class Message
229 {
        ... 中略 ...
342     int numberOfHeaders;
343     int headerProtocols[MAX_HEADERS];
344     int headerSizes[MAX_HEADERS];
        ... 中略 ...
350     std::vector<MessageInfoHeader> infoArray;
351     std::vector<MessageInfoBookKeeping> infoBookKeeping;
        ... 中略 ...
426 };
```

Verctor と配列の 2 種類の管理であるから、実際のメモリ領域をアクセスする場合には正しい方法でアクセスしなければならない。特に配列で管理されている header は添え字の値に注意が必要である。

2.3.6.5 Message 本体の具体的なデータ領域構造

Message 本体におけるデータ領域には、info 領域、header 領域、packet 領域の 3 つがある。

message.h

```
228 class Message
229 {
        ... 中略 ...
313     char *packet;
        ... 中略 ...
318     char *payload;
        ... 中略 ...
342     int numberOfHeaders;
343     int headerProtocols[MAX_HEADERS];
344     int headerSizes[MAX_HEADERS];
        ... 中略 ...
350     std::vector<MessageInfoHeader> infoArray;
351     std::vector<MessageInfoBookKeeping> infoBookKeeping;
        ... 中略 ...

426 };
```

Message にはパケットのデータを管理するポインタとして packet と payload の 2 つがある。
この 2 つの実体を伴う領域は payload から payloadSize の連続領域としてメモリ確保され、この payload 領域の中に packet の領域が含まれている。

packetSize は PHY レイヤで使用されるデータの byte 数を表す。これは実体を伴うデータ(packet)と実体を伴わないデータ(virtualPayload)を含んだものになる。
packetSize は PHY レイヤにおける伝搬すべきデータのサイズとして使用され、PHY におけるデータレート(bps)で計算することで、データ伝搬に必要な時間が求められる。

payloadSize は QualNet 内部で使用されるデータの byte 数を表す。これは、QualNet 内部における Message 管理領域(MSG_MAX_HDR_SIZE (512 固定で定義)byte が加算されたサイズとなる。

これらを踏まえて実際の Messase 構造を図化すると以下のようになる。

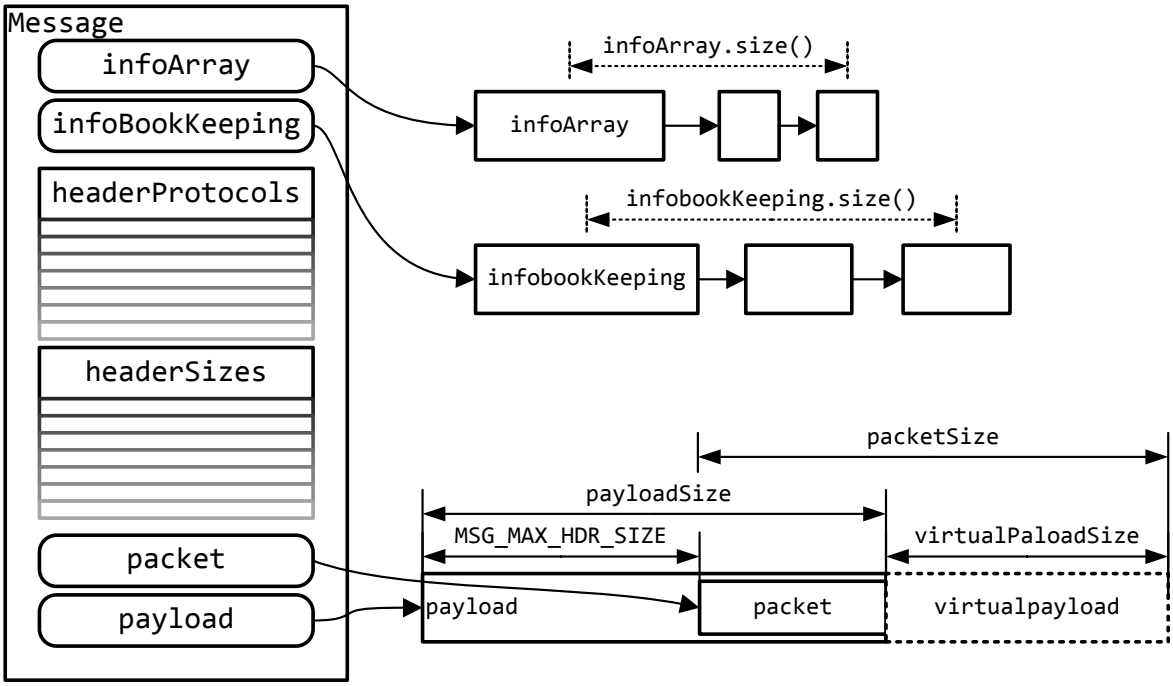

図 6　Message クラス構造

2.3.6.6　payload 領域管理

実は payload 領域も Message 本体と同じように QualNet 内部処理を高速化するためにメモリプール管理（cache）が行われている。
MESSAGE_NO_RECYCLE の定義と密接に関連しているが、
MAX_CACHED_PAYLOAD_SIZE(1024 で定義)よりも大きなサイズは無条件に MEM_Alloc される。
また、MESSAGE_NO_RECYCLE されない(つまり RECYCLE される)場合、Message 本体とは別に再利用され、その個数は MSG_PAYLOAD_LIST_MAX(1000 で定義)である。

【MESSAGE_PayloadAlloc 関数の実態】
Cache 処理のためにやや複雑な処理になっている。

message.cpp

```
1694 char* MESSAGE_PayloadAlloc(PartitionData *partition,
1695                            int payloadSize,
1696                            bool isMT)
1697 {
1698     if ((payloadSize/* + MSG_MAX_HDR_SIZE*/) > MAX_CACHED_PAYLOAD_SIZE)
1699     {
1700         return (char *) MEM_malloc(payloadSize/* + MSG_MAX_HDR_SIZE*/);
1701     }
1702     else
1703     {
1704 #ifdef MESSAGE_NO_RECYCLE
1705         char *ptr = (char *) MEM_malloc(payloadSize/* + MSG_MAX_HDR_SIZE*/);
1706         memset(ptr, 0, payloadSize);
1707         return ptr;
1708 #else
1709         if ((partition->msgPayloadFreeList == NULL) || (isMT))
1710         {
1711             MessagePayloadListCell* NewCell = (MessagePayloadListCell*)
1712                 MEM_malloc(sizeof(MessagePayloadListCell));
1713             return (char *) &(NewCell->payloadMemory[0]);
1714         }
1715         else
1716         {
1717             char *payload = (char *)
1718                 &(partition->msgPayloadFreeList->payloadMemory[0]);
1719             partition->msgPayloadFreeList =
1720                 partition->msgPayloadFreeList->next;
1721             (partition->msgPayloadFreeListNum)--;
1722
1723             return payload;
1724         }
1725 #endif
1726     }
1727 }
```

【MESSAGE_PayloadFree 関数の実態】

message.cpp

```
1755 void MESSAGE_PayloadFree(PartitionData *partition,
1756                           char *payload,
1757                           int payloadSize,
1758                           bool wasMT)
1759 {
1760 #ifdef MESSAGE_NO_RECYCLE
1761     MEM_free(payload);
1762 #else
1763     if ((partition != NULL) &&
1764         (payloadSize <= MAX_CACHED_PAYLOAD_SIZE) &&
1765         (wasMT == false) &&
1766         (partition->msgPayloadFreeListNum < MSG_PAYLOAD_LIST_MAX))
1767     {
1768         MessagePayloadListCell* cellPtr =
1769             (MessagePayloadListCell*)payload;
1770         cellPtr->next = partition->msgPayloadFreeList;
1771         partition->msgPayloadFreeList = cellPtr;
1772         (partition->msgPayloadFreeListNum)++;
1773     }
1774     else
1775     {
1776         MEM_free(payload);
1777     }
1778 #endif
1779 }
```

2.3.6.7 header 領域管理

payload と異なり header は配列で管理されているため、領域管理は単純である。

Message の割当と解放の場合とは異なり、MESSAGE_HeaderAlloc と MESSAGE_HeaderFree という関数名ではなく、MESSAGE_AddHeader と MESSAGE_RemoveHeader という関数名なっているので注意する必要がある。さらに注意すべきことに、header 領域は配列で MAX_HEADERS(10 と定義) 個確保されているが、実際の処理では MAX_HEADERS を超えたか否かのチェックを行っていない。もし、10 個よりも多い header が必要となる場合は(たぶん 10 個で足りるとは思うが)、何らかの対処が必要となる。

【MESSAGE_AddHeader 関数の実態】
1408 行目の次で numberOfHeaders の値と MAX_HEADERS の値をチェックすべきかもしれない。

message.cpp

```
1394 void MESSAGE_AddHeader(Node *node,
1395                         Message *msg,
1396                         int hdrSize,
1397                         TraceProtocolType traceProtocol)
1398 {
1399     msg->packet -= hdrSize;
1400     msg->packetSize += hdrSize;
1401
1402     if (msg->isPacked)
1403     {
1404         msg->actualPktSize += hdrSize;
1405     }
1406     msg->headerProtocols[msg->numberOfHeaders] = traceProtocol;
1407     msg->headerSizes[msg->numberOfHeaders] = hdrSize;
1408     msg->numberOfHeaders++;
1409
1410     if (msg->packet < msg->payload) {
1411         char errorStr[MAX_STRING_LENGTH];
1412         sprintf(errorStr, "Not enough space for headers.\n"
1413                 "Increase the value of MSG_MAX_HDR_SIZE and try again.\n");
1414         ERROR_ReportError(errorStr);
1415     }
1416 }
```

【MESSAGE_RemoveInfo 関数の実態】
1447 行の次で numberOfHeaders の値がマイナスにならないことをチェックすべきかもしれない。

message.cpp

```
1431 void MESSAGE_RemoveHeader(Node *node,
1432                           Message *msg,
1433                           int hdrSize,
1434                           TraceProtocolType traceProtocol)
1435 {
1436     ERROR_Assert(msg->headerProtocols[msg->numberOfHeaders-1] == traceProtocol,
1437                  "TRACE: Removing trace header that doesn't match!\n");
1438
1439     msg->packet += hdrSize;
1440     msg->packetSize -= hdrSize;
1441
1442     if (msg->isPacked)
1443     {
1444         msg->actualPktSize -= hdrSize;
1445     }
1446
1447     msg->numberOfHeaders--;
1448
1449     if (msg->packet > (msg->payload + msg->payloadSize)) {
1450         char errorStr[MAX_STRING_LENGTH];
```

```
1451         sprintf(errorStr, "Packet pointer going beyond allocated memory.\n");
1452         ERROR_ReportError(errorStr);
1453     }
1454 }
```

2.3.6.8 info 領域管理

info 領域の管理は一番ややこしい。

関数は Header と同様に MESSAGE_AddInfo と MESSAGE_RemoveInfo であるが、子供の関数として MESSAGE_InfoFiledAlloc と MESSAGE_InfoFiledFree も使う。

【MESSAGE_AddInfo 関数の実態】

Info 領域は Vector で管理されているため、Add と Remove では Vector として処理する。

message.cpp

```
741 char* MESSAGE_AddInfo(PartitionData *partition,
742                         Message *msg,
743                         int infoSize,
744                         unsigned short infoType)
745 {
746     MessageInfoHeader infoHdr;
747     MessageInfoHeader* hdrPtrInfo = NULL;
748     bool insertInfo = false;
749     unsigned int i;
750
751     ERROR_Assert(infoSize != 0, "Cannot add empty info");
752
753     if (infoType == INFO_TYPE_DEFAULT)
754     {
755         // First check if there is already a default type
756         if (msg->infoArray.size() > 0)
757         {
758             for (i = 0; i < msg->infoArray.size(); i++)
759             {
760                 MessageInfoHeader* info;
761                 info = &(msg->infoArray[i]);
762                 if (info->infoType == infoType)
763                 {
764                     hdrPtrInfo = &(msg->infoArray[i]);
765                     break;
766                 }
767             }
768         }
769
770         if (hdrPtrInfo == NULL)
771         {
772             // There is no default info so we will insert it
773             insertInfo = true;
774
775             // Use small info space if info is small enough.
776             if (infoSize <= SMALL_INFO_SPACE_SIZE)
777             {
778                 infoHdr.info = (char*) msg->smallInfoSpace;
779             }
780             else
781             {
782                 // TODO: Check that this is mem_freed
783                 infoHdr.info = (char*) MEM_malloc(infoSize);
784                 memset(infoHdr.info, 0, infoSize);
785             }
786
787             infoHdr.infoSize = infoSize;
788             infoHdr.infoType = INFO_TYPE_DEFAULT;
```

```
789         }
790         else
791         {
792             // There is already a default info in the list.  Reuse it.
793             if (infoSize <= SMALL_INFO_SPACE_SIZE)
794             {
795                 // Free old memory if it was not using small info space
796                 if (hdrPtrInfo->infoSize > SMALL_INFO_SPACE_SIZE)
797                 {
798                     MEM_free(hdrPtrInfo->info);
799                 }
800
801                 hdrPtrInfo->info = (char*) msg->smallInfoSpace;
802                 memset(hdrPtrInfo->info, 0, SMALL_INFO_SPACE_SIZE);
803             }
804             else
805             {
806                 // Free old memory if new info is bigger, otherwise reuse it.
807                 if ((unsigned int)infoSize > hdrPtrInfo->infoSize
808                     && hdrPtrInfo->infoSize > SMALL_INFO_SPACE_SIZE)
809                 {
810                     // Old one was not small info, free then allocate
811                     MEM_free(hdrPtrInfo->info);
812                     hdrPtrInfo->info = (char*) MEM_malloc(infoSize);
813                 }
814                 else if ((unsigned int)infoSize > hdrPtrInfo->infoSize)
815                 {
816                     // Old one was small info, allocate new memory
817                     hdrPtrInfo->info = (char*) MEM_malloc(infoSize);
818                 }
819
820                 memset(hdrPtrInfo->info, 0, infoSize);
821             }
822
823             hdrPtrInfo->infoSize = infoSize;
824         }
825     }
826     else
827     {
828         // Info type is not default
829
830         // Loop over all infos.  Use INFO_TYPE_UNDEFINED unless we find
831         // an exact infoType match.
832         for (i = 0; i < msg->infoArray.size(); i++)
833         {
834             if (msg->infoArray[i].infoType == infoType)
835             {
836                 hdrPtrInfo = &(msg->infoArray[i]);
837                 memset(hdrPtrInfo->info, 0, hdrPtrInfo->infoSize);
838                 break;
839             }
840             else if (hdrPtrInfo == NULL &&
841                      msg->infoArray[i].infoType == INFO_TYPE_UNDEFINED)
842             {
843                 hdrPtrInfo = &(msg->infoArray[i]);
844                 memset(hdrPtrInfo->info, 0, hdrPtrInfo->infoSize);
845             }
846         }
847
848         if (hdrPtrInfo == NULL)
849         {
850             // No info was found.  We will insert infoHdr.
851             insertInfo = true;
852
853             infoHdr.info = MESSAGE_InfoFieldAlloc(partition,
854                                                   infoSize,
855                                                   msg->mtWasMT);
856             memset(infoHdr.info, 0, infoSize);
857             infoHdr.infoSize = infoSize;
```

```
858             infoHdr.infoType = infoType;
859         }
860         else
861         {
862             // Info was found.  Enlarge if either new or old infos are bigger
than
863             // small info space.  Then set the type.
864             if (hdrPtrInfo->infoSize < (unsigned int)infoSize &&
865                 (hdrPtrInfo->infoSize > SMALL_INFO_SPACE_SIZE ||
866                 infoSize > SMALL_INFO_SPACE_SIZE))
867             {
868                 if (hdrPtrInfo->infoSize > 0)
869                 {
870                     MESSAGE_InfoFieldFree(partition,
871                                           hdrPtrInfo,
872                                           msg->mtWasMT);
873                 }
874
875                 hdrPtrInfo->info = MESSAGE_InfoFieldAlloc(partition,
876                                                           infoSize,
877                                                           msg->mtWasMT);
878                 hdrPtrInfo->infoSize = infoSize;
879                 memset(hdrPtrInfo->info, 0, infoSize);
880             }
881
882             hdrPtrInfo->infoType = infoType;
883         }
884     }
885
886     // If we are adding new info push it onto the array
887     if (insertInfo)
888     {
889         msg->infoArray.push_back(infoHdr);
890         return infoHdr.info;
891     }
892
893     // We reused old info
894     return hdrPtrInfo->info;
895 }
```

厄介なことに、Info 領域も cache される。一度使って不要になった Info 領域はすぐに解放せず、次に割り当てる場合に必要なサイズをチェックし、再利用（サイズが小さい）可能であれば再利用する。

【MESSAGE_RemoveHeader 関数の実態】
Add と比較すると Remove は比較的単純な処理になっている。

message.cpp

```
909 void MESSAGE_RemoveInfo(Node *node, Message *msg, unsigned short infoType)
910 {
911     unsigned int i;
912     MessageInfoHeader* hdrPtrInfo = NULL;
913
914     // Remove the info field from the vector info Array.
915     for (i = 0; i < msg->infoArray.size(); i ++)
916     {
917         hdrPtrInfo = &(msg->infoArray[i]);
918         if (infoType == hdrPtrInfo->infoType)
919         {
920             break;
921         }
922     }
923
924     if (i < msg->infoArray.size())
925     {
926         if (hdrPtrInfo->infoSize > 0)
927         {
```

```
928             MESSAGE_InfoFieldFree(node, hdrPtrInfo);
929         }
930
931         hdrPtrInfo->infoType = INFO_TYPE_UNDEFINED;
932         hdrPtrInfo->infoSize = 0;
933         hdrPtrInfo->info = NULL;
934
935         // Remove the entry from the vector.
936         msg->infoArray.erase(msg->infoArray.begin()+i);
937     }
938 }
```

このように QualNet の内部では Message 本体だけでなく、Message が管理している領域も cache 処理が行われている。Message 処理は QualNet 本体の処理で一番利用頻度が多い処理であるから、極力処理を高速化しておかなければならない。そのために一見複雑な cache 処理が行われているが、QualNet を高速に実行するためには必須の処理である。

一般的にメモリ管理はコスト(計算時間)が高い。メモリは OS が全体管理しているので、QualNet という一つのプログラムが自由に使うことはできない。必ず OS に境域を要求し、不要になったら OS に領域を返す。OS に要求を出すということは OS 内部で処理が実行されることになりコストが高くつく。従って、できる限り QualNet 内部でメモリの再利用(cache)を行っている。

【コラム】

> 実際には、malloc と free はライブラリであり OS の内部処理ではない。malloc や free も自前の cache 処理を行い、実際の OS にメモリを要求する処理を必要最小限にとどめている。
> Windows(Microsoft VC ライブラリ)と Linux(glibc)では、malloc や free の cache アルゴリズムが異なる。これが影響して、大量の Message を使用するシナリオを QualNet で実行すると、同一のハードウェアを使用していても Windows と Linux で QualNet の実行速度が異なる場合がある。

2.4 イベント処理の実装

2.4.1 イベント処理の概要

ここまでのおさらいになるが、離散事象型ネットワークシミュレータである QualNet は、シミュレーションの対象をイベント(discrete event)を駆使してモデル化する。イベントはプログラム上では Message というクラス構造を用いて表現される。従って、QualNet においては「メッセージ」という語は多くの文脈において「イベント」と同じ意味で使われる。

QualNet が扱うメッセージには、大きく分けてパケットとタイマがある。パケットはネットワークにおけるモデル化対象のエンティティ(つまりパケット)を直接表現する目的で用意されている。それに対して、タイマはよりプリミティブな概念であり、通常、シミュレーションモデルの時間軸上の振る舞いや処理を実現するのに用いられる。以下では、これらを区別するために、パケットを表すメッセージのことをパケットメッセージ、タイマを表すメッセージのことをタイマメッセージと必要に応じて呼び分けることにする。

それでは、タイマメッセージとパケットメッセージの使い方について以下で詳しく見ていく。

2.4.2 タイマの使い方

離散事象シミュレーションにおいてタイマを使う局面を考えたとき、代表的な使い方は以下の二種類のいずれかとなる。

(ア) 繰り返し処理の実現（Interval Timer）
(イ) 期限の設定とタイムアウト処理の実現（Single Shot Timer）

以下、それぞれについて解説する。

2.4.2.1 タイマを用いた繰り返し処理（Interval Timer）の実現

まず、一般に Interval Timer と呼ばれる類のタイマの使い方である「繰り返し処理の実現」について説明する。通常の逐次処理プログラミングにおいて、繰り返し処理を実現しようとした場合には、以下に示すようにプログラミング言語のループ制御構造を使って記述するのが一般的である。

```
    // 1 秒の間隔を空けて 100 回パケットを送信する。
    for (int i = 0; i < 100; i ++)
    {
        SendPacket(pkt, dstAddr);
        sleep(1);
    }
```

しかしながら、離散事象シミュレーションのプログラムで同じ記述をしても、おそらく期待した結果は得られない。なぜなら、"sleep(1)"のようなおそらくシステムコールによるスリープ処理は、プログラムの動作を実時間上で 1 秒だけ遅らせるが、シミュレーション時間の進捗には全く影響を与えないからである。離散事象シミュレーションの世界は、計算機がプログラムコードを 1 ステップずつ実行する際に費やされる実時間(real time)とは全く別次元の、シミュレーション時間(simulation time)が支配する特殊な世界であることを念頭に入れておく必要がある。シミュレーション時間を進めるには、未来のイベントをスケジュールすればよい。これが**タイマ**である。

QualNet で上記の繰り返し処理を実現しようとする場合には、初期化処理とタイマ発火時の処理において以下のようにプログラムを記述する。(以下では、仮に、対象プロトコル種別を *PROTOCOL*、対象プロトコルのデータ構造を *PROTOCOL_DATA*、対象プロトコルが属するレイヤを *LAYER*、仕掛けるタイマのメッセージ種別を *MSG_TYPE* という前提で記述している。)

- 初期化処理において、最初のパケット送信処理を行い、パケット送信回数を記録した上で、さらに 1 秒後に発火するタイマを仕掛ける。

```
PROTOCOL_Initialize(NODE *node, ... )
{
    Message *timerMsg;
    ... 中略 ...

    // 最初のパケットを送信する
    SendPacket(pkt, dstAddr);

    // パケット送信回数を記録する
    node->PROTOCOL_DATA->numPktSent++;

    // タイマメッセージを作成する
    timerMsg = MESSAGE_Alloc(node, LAYER, PROTOCOL, MSG_TYPE)

    // タイマメッセージを構築する
    ... 中略 ...

    // 1 秒後に発火するタイマを仕掛ける
```

```
    MESSAGE_Send(node, timerMsg, 1 * SECOND);
}
```

- 仕掛けたタイマの発火時の処理において、パケット送信回数をチェックし、100 回に満たなければ初期化時と同様にパケットの送信、パケット送信回数の記録、タイマの登録を行う。

```
PROTOCOL_ProcessEvent(NODE *node, Message *msg, ... )
{
    Message *timerMsg;
    ... 中略 ...

    if (node->PROTOCOL_DATA->numPktSent < 100)
    {
        // パケットを送信する
        SendPacket(pkt, dstAddr);

        // パケット送信回数を記録する
        node->PROTOCOL_DATA->numPktSent++;

        // タイマメッセージを作成する
        timerMsg = MESSAGE_Alloc(node, LAYER, PROTOCOL, MSG_TYPE)

        // タイマメッセージを構築する
        ... 中略 ...

        // 1 秒後に発火するタイマを仕掛ける
        MESSAGE_Send(node, timerMsg, 1 * SECOND);
    }

    // 今回配送されたタイマメッセージはもう不要なので解放する
    MESSAGE_Free(node, msg);
}
```

なお、上記の「タイマを仕掛ける」という処理は、サンプルコードを見て分かる通り、イベントスケジューラにタイマイベントを登録（タイマメッセージを送信）することを意味している。

以上の処理の全体像を図 7 に示す。タイマメッセージは、イベントスケジューラが NODE_ProcessEvent 関数を呼び出すことで対象ノードに配送され、対象ノード内で対象レイヤのイベント処理関数が呼び出されさらに対象プロトコルのイベント処理関数(ここでは *PROTOCOL*_ProcessEvent 関数)が呼び出されるようになっている。

一般的なプログラミングと比べて少々面倒だが、離散事象シミュレーションのプログラミングにおいては、繰り返し処理をこのような手順で記述する必要がある。

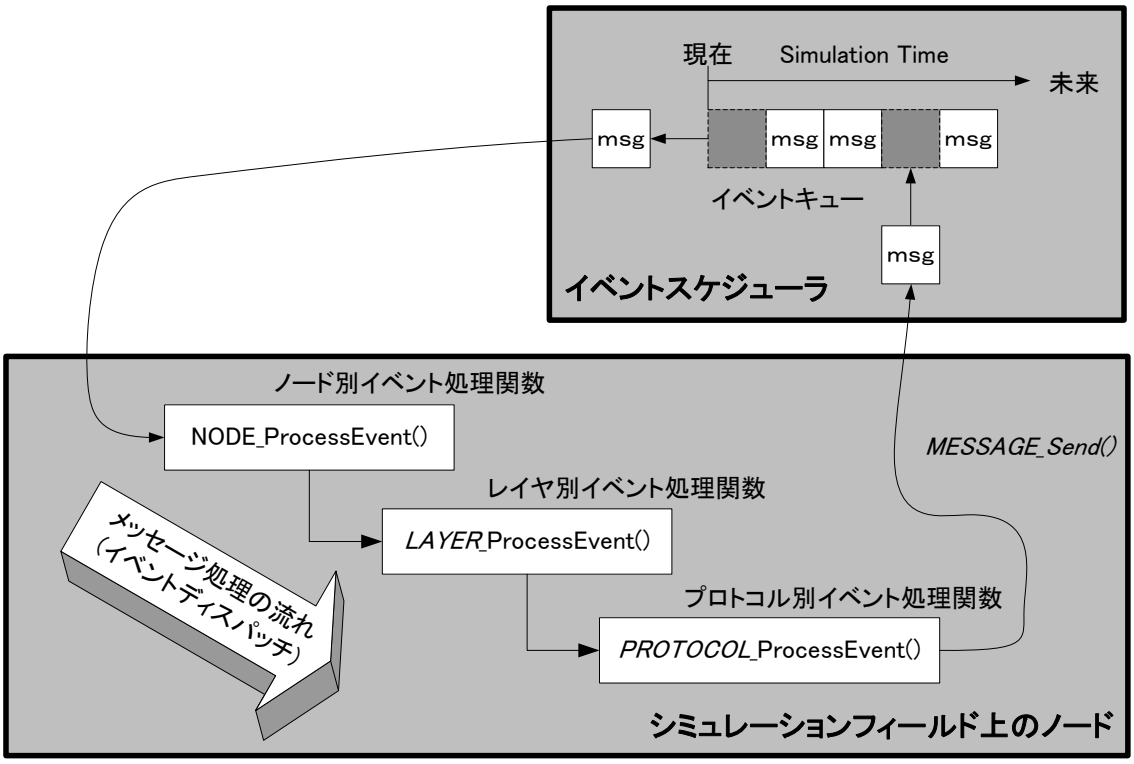

図 7　QualNet 上で繰り返し処理が実行される様子

2.4.2.2　期限の設定とタイムアウト処理（Single Shot Timer）の実現

もう一つの代表的な例は、一般に Single Shot Timer と呼ばれる類のタイマの使い方で、期限の設定とタイムアウト処理の実現である。ネットワークシミュレーションの場合、例えば要求パケットを送信した後に一定期間内に応答パケットが返って来たかどうかを判断する場合などに使われる。複雑なプロトコルモデルになると、レイヤ毎に複数のタイマが同時に仕掛けられることもある。

上記に例を挙げた要求パケット送信とその応答パケット待ちのタイムアウト処理を実現したい場合には、要求パケット送信時、応答待ちタイムアウトタイマ発火時、および応答パケット受信時の処理において以下のようにプログラムを記述する。以下、応答待ちタイムアウトまでの時間は仮に 1 秒とする。

- 要求パケット送信処理において、要求パケットの送信を行い、応答待ちタイムアウトのためのタイマを仕掛け、後のキャンセル処理時のために仕掛けたタイマメッセージへのポインタを記録しておく。

```
PROTOCOL_SendRequest(NODE *node, ... )
{
    Message *timerMsg;
    ... 中略 ...

    // 要求パケットを送信する
    SendPacket(pkt, dstAddr);

    // タイマメッセージを作成する
    timerMsg = MESSAGE_Alloc(node, LAYER, PROTOCOL, MSG_TYPE)

    // タイマメッセージを構築する
    ... 中略 ...

    // 1 秒後に発火する応答待ちタイムアウトタイマを仕掛ける
    MESSAGE_Send(node, timerMsg, 1 * SECOND);
```

```
    // 仕掛けたタイマメッセージへのポインタを記録しておく
    node->PROTOCOL_DATA->replyWaitTimer = timerMsg;
}
```

- 応答待ちタイムアウトタイマ発火時の処理においては、expire したタイマメッセージへのポインタの記録が残ったままになっているので、それをクリアする。

```
PROTOCOL_ProcessEvent(NODE *node, Message *msg, ...)
{
    // タイムアウト処理を行う(User-Oriented な処理)
    // 例えばプロトコルリセット処理や要求の再送処理など
    ... 中略 ...

    // expire したタイマメッセージへのポインタをクリアしておく
    node->PROTOCOL_DATA->replyWaitTimer = NULL;
}
```

- 応答パケット受信処理において、この時点でまだ要求パケット送信時に仕掛けたタイマは expire していない(つまりイベントキューに残っている)はずだから、タイマのキャンセル処理を行う。キャンセル処理はタイマをキューから取り除くのではなく、ただ"cancel フラグ"を立てるだけである。このときメッセージはキューに残ったままなので、決して MESSAGE_Free 関数を呼び出して解放(メモリプールに返却)してはならない。cancel フラグの立ったメッセージは、イベントスケジューラによって自動的に解放(メモリプールに返却)されるようになっているので、ユーザコード側ではフラグを立てるだけで後の処理はイベントスケジューラに任せればよい。万一 MESSAGE_Free 関数を呼び出して解放(メモリプールに返却)してしまうと、誰かがその領域を再利用してしまうため、メモリ領域の破壊が起こりデバッグは困難になる。キャンセルしたメッセージをキューから取り除かないのは、イベントスケジューラに任せる場合と比べて処理負荷が高いからである。

```
PROTOCOL_RecvReply(NODE *node, ... )
{
    // パケット受信処理を行う(User-Oriented な処理)
    ... 中略 ...

    // 仕掛けたタイマメッセージをキャンセルする
    MESSAGE_CancelSelfMsg(node, node->PROTOCOL_DATA->replyWaitTimer);
    // MESSAGE_Free(node, node->PROTOCOL_DATA->replyWaitTimer); //これはやったらダメ！
    node->PROTOCOL_DATA->replyWaitTimer = NULL; //これはやったほうがよい！
}
```

2.4.2.3 サンプル実装コード

以下では、実例として CBR アプリケーションで行われている定期タイマ処理の説明を行う。

【メッセージの作成と送信】

CBR クライアントは、CBR パケットを定期送信するためにタイマメッセージを利用する。1 パケット送信すると、次の定期送信時刻に発火するタイマを設定し、そのタイマが発火するとまた 1 パケット送信する、ということを繰り返す。以下は、次回定期送信時刻に発火するタイマをセットしているコードである。

app_cbr.cpp

```
966 void //inline//
 967 AppCbrClientScheduleNextPkt(Node *node, AppDataCbrClient *clientPtr)
 968 {
 969     AppTimer *timer;
 970     Message *timerMsg;
```

```
 971
 972     timerMsg = MESSAGE_Alloc(node,
 973                              APP_LAYER,
 974                              APP_CBR_CLIENT,
 975                              MSG_APP_TimerExpired);
 976
 977     MESSAGE_InfoAlloc(node, timerMsg, sizeof(AppTimer));
 978
 979     timer = (AppTimer *)MESSAGE_ReturnInfo(timerMsg);
 980     timer->sourcePort = clientPtr->sourcePort;
 981     timer->type = APP_TIMER_SEND_PKT;

         ... 中略 ...

1000     MESSAGE_Send(node, timerMsg, clientPtr->interval);
1001 }
```

メッセージの作成と送信

行番号	処理内容
972-975	タイマメッセージの生成。引数として、送信者である自身と同じ APP_LAYER、APP_CBR_CLIENT を与えている。第 4 引数には、アプリケーション層で使われるタイマであることを表す MSG_APP_TimerExpired を与えている。これは QualNet 全体でメッセージの種別を特定するための定数で、api.h で定義されている。
977-981	生成したメッセージに対する INFO の付与。付加情報として AppTimer を含ませて、各種情報を格納している。ここでは、タイマの種別と CBR クライアントを特定するための送信元ポートを格納している。
1000	生成したメッセージの送信。第 3 引数にタイマが発火するまでの時間を与える。この例では、CBR パケットを次に送信すべき時刻までの時間、すなわち定期送信間隔を与えている。

【メッセージの配送】

送信されたメッセージは、指定のレイヤ種別、プロトコル種別を元に、CBR クライアントまで配送される。

node.cpp

```
225 void
226 NODE_ProcessEvent(Node *node, Message *msg)
227 {
        ... 中略 ...

271     switch (MESSAGE_GetLayer(msg))
272     {
            ... 中略 ...
303         case APP_LAYER:
304         {
305             APP_ProcessEvent(node, msg);
306             break;
307         }
            ... 中略 ...
363     }
364
365     SimContext::unsetCurrentNode();
366 }
```

application.cpp

```
5844 void APP_ProcessEvent(Node *node, Message *msg)
5845 {
         ... 中略 ...
5883     switch(protocolType)
5884     {
```

```
            ... 中略 ...
5974        case APP_CBR_CLIENT:
5975        {
5976            AppLayerCbrClient(node, msg);
5977            break;
5978        }
            ... 中略 ...
6358    }//switch//
6359 }
```

メッセージの配送

行番号	処理内容
226	設定時刻になったメッセージはまずこの NODE_ProcessEvent で該当ノードに配送される。
303-307	レイヤの種別 APP_LAYER を元にアプリケーション層のイベント処理関数 APP_ProcessEvent が呼び出される。
5974-5978	プロトコルの種別 APP_CBR_CLIENT を元に CBR クライアントのイベント処理関数 AppLayerCbrClient が呼び出される。

【メッセージの受信と破棄】

メッセージ受信したら必要処理を行い、メッセージを破棄する。必要があれば付与されている INFO を取り出す。

app_cbr.cpp

```
143 void
144 AppLayerCbrClient(Node *node, Message *msg)
145 {
        ... 中略 ...
156     switch (msg->eventType)
157     {
158         case MSG_APP_TimerExpired:
159         {
160             AppTimer *timer;
161
162             timer = (AppTimer *) MESSAGE_ReturnInfo(msg);
                ... 中略 ...
169             clientPtr = AppCbrClientGetCbrClient(node, timer->sourcePort);
                ... 中略 ...
179             switch (timer->type)
180             {
181                 case APP_TIMER_SEND_PKT:
182                 {
                        ... 中略 ...
297                 }
                    ... 中略 ...
305             }
306
307             break;
308         }
            ... 中略 ...
317     }
318
319     MESSAGE_Free(node, msg);
320 }
```

メッセージの受信と破棄

行番号	処理内容
156-158	メッセージ自体の種別によって処理を振り分けている。この case 文はタイマメッセージを

処理するブロックである。

162　送信時に格納した INFO 領域から、AppTimer を取り出す。

169　AppTimer に格納されている送信元ポート番号をここで使用している。タイマに対応した(今定期送信すべき)CBR クライアントを特定している。

179-181　タイマの種別によって処理を振り分けている。この case 文は CBR パケット定期送信用のタイマ発火時に起動されるブロックである。

319　不要になったタイマメッセージを解放する。

なお、以上の例では、レイヤ種別 - プロトコル種別 - タイマ種別という階層でタイマメッセージを扱っている例を示したが、この他にプロトコル種別自体に複数タイマそれぞれに対応した値を定義することでタイマメッセージを扱う方法もある。例えば以下の mac_dot11 モデルの例では、レイヤ種別、プロトコル種別により配送されて来たメッセージを非タイマメッセージと同階層の switch 文で処理分岐させていることが確認できる。

mac_dot11.cpp

```
2657 void MacDot11Layer(Node* node, int interfaceIndex, Message* msg)
2658 {
          ... 中略 ...
2665      switch (msg->eventType) {
2666          case MSG_MAC_TimerExpired: {
                  ... 中略 ...
2684              MESSAGE_Free(node, msg);
2685              break;
2686          }
              ... 中略 ...
2689          case MSG_MAC_DOT11_Beacon: {
                  ... 中略 ...
2739              MESSAGE_Free(node, msg);
2740              break;
2741          }
              ... 中略 ...
2866          case MSG_MAC_DOT11_Station_Inactive_Timer:
2867          {
                  ... 中略 ...
2916             MESSAGE_Free(node, msg);
2917             break;
2918          }
              ... 中略 ...
2975      } //switch
2976
2977      // dot11s. Some mesh messages reuse timers.
2978      //MESSAGE_Free(node, msg);
2979 }//MacDot11Layer
```

2.4.3 パケットの使い方

パケットはシミュレーションモデルの処理の流れを記述するために便宜的に使われるタイマと異なり、モデル化対象の実在するエンティティ(パケット)を表すのに用いられる。パケットと言えども離散事象シミュレーションの世界では、イベントの一形態として表現することに変わりはないのだが、パケットにはそれ自身が本来持っている性質がある。ネットワークシミュレータである QualNet は、パケットという特別なイベントの性質をモデル化するための機構を、予め豊富に取り揃えている。

2.4.3.1 タイマとパケットの違い

タイマとパケットの最も大きな違いは、パケットがモデル化対象の(リアルな) 世界において、データを運ぶエンティティであるという点である。図 8(右図) に示すように、パケットはそのために特別な「packet フィールド」を Message 内で与えられている。図 8(左図) に示したタイマメッセージもよく似た「info フィールド」を複数持つことができるので、タイマメッセージを使ってパケットメッセージを表現することももちろん可能である。

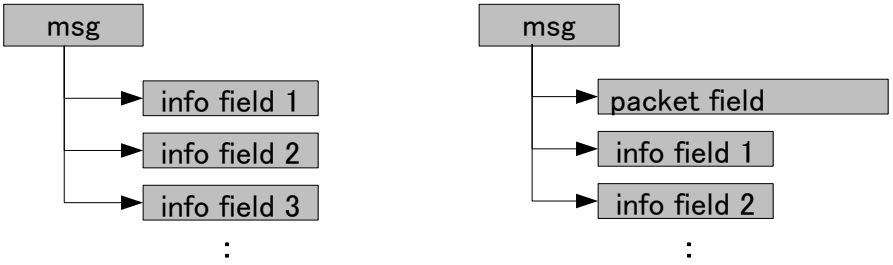

図 8 タイマ(左図)とパケット(右図)

QualNet はネットワークシミュレータとして出発した経緯から、より適用領域に近い「パケット」がタイマとは別に存在し、パケットを表現するための様々な API 関数群が用意されている。それら API 関数群を用いて行われるパケットに対する操作のうち代表的なものを以下で説明する。

2.4.3.2 レイヤ間でのパケットのやり取り

図 9 は QualNet においてあるノード上のアプリケーション層が送信したパケットを、別のノードのアプリケーション層が受信するまでの流れを概念的に図示したものである。

送信処理(図の左半分)において、アプリケーション層がパケットを生成した後、下位レイヤであるトランスポート層にそのパケットを送信している。トランスポート層は IP 層に、IP 層はルーティング層とやり取りしながらそのパケットを MAC 層に送信する。その際、各レイヤはパケットの先頭にヘッダを付加していく。

受信処理(図の右半分)においては、PHY 層で信号として受信されたパケットが次々と上位レイヤに渡されていく中で、各レイヤにて不要となったヘッダを除去され、最終的にアプリケーション層に送信側アプリケーション層が生成したものと同じデータパケットが届けられる。

送信側 PHY 層まで下りてきたパケットメッセージは、図には記されていないがその下位に存在する Propagation レイヤに信号送信イベントとして通知される。Propagation レイヤは信号受信イベントを受信側 PHY 層に通知する。このノード間における信号のやり取りについては後ほど詳しく説明する。

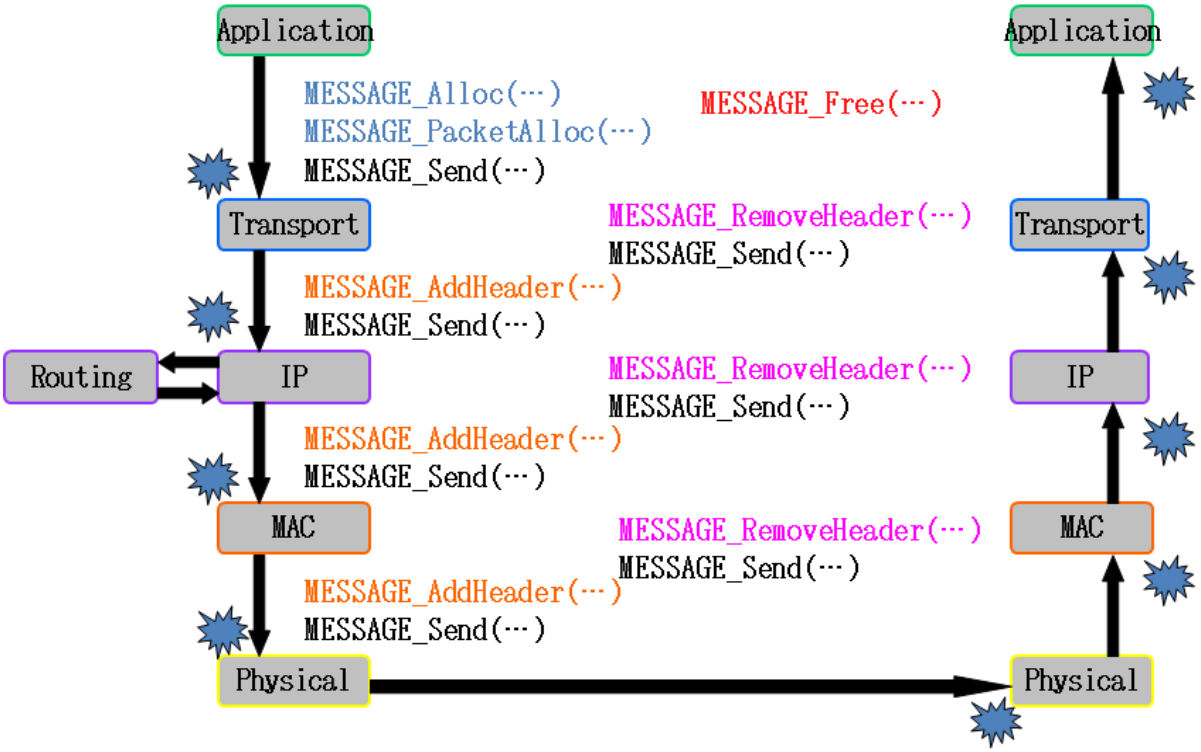

図 9　メッセージ処理の概要（パケットの流れ）

なお、実際のコード上は必ずしも上図の通りになっていないので注意する必要がある。実際、アプリケーション層-トランスポート層間以外のレイヤ間でのパケット受け渡しは MESSAGE_Send 関数を用いてイベントスケジューラ経由で行うことは稀であり、ほとんどの場合イベントスケジューラを介さずに直接パケットメッセージの受け渡しを行っている。例えば TCP や UDP が IP 層にパケットを渡す場合は、IP 層が用意している NetworkIpReceivePacketFromTransportLayer 関数を呼び出して直接パケットメッセージを引き渡しているし、また IP 層が MAC 層にパケットを渡す場合は、インタフェースキューにパケットメッセージを格納した上で MAC_NetworkLayerHasPacketToSend 関数を呼び出して MAC 層プロトコルにその旨を伝えている。このような直接引き渡しの場合、呼び出し側レイヤにおけるパケット処理遅延はないものという前提で実装されていることは理解しておく必要がある。逆に、例えば IP パケット処理でルータバックプレーンを経由させる場合など(NetworkIpUseBackplaneIfPossible 関数)、パケット処理遅延を意図的に入れたい箇所では、MESSAGE_Send 関数を用いて指定時間後に指定レイヤにパケットメッセージが届けられるように実装されている。

2.4.3.3　ヘッダの追加と削除(MESSAGE_AddHeader 関数と MESSAGE_RemoveHeader 関数)

パケット交換ネットワークの世界では、パケットの送信元や宛先のネットワークアドレスを、ヘッダという形でパケットに付加することで、パケットが正しい宛先に届けられることを保証する。ヘッダはレイヤ毎に存在し、例えばトランスポートレイヤの TCP は TCP ヘッダを、ネットワークレイヤの IP は IP ヘッダを付加する。従って、アプリケーションレイヤからパケットがレイヤを一つずつ降りていく度に、ヘッダは次々とパケット(ペイロード)の先頭に付加されてパケットサイズが大きくなっていく。図 10(左図)はその様子を図式化したものである。逆にパケットの受信処理の場合、下位レイヤからパケットが一つずつレイヤを上がってく

る時、不要になったヘッダは各レイヤにて削除され、ヘッダの取り除かれたペイロード部分だけが上位レイヤに渡される。図 10(右図)はその様子を図式化している。

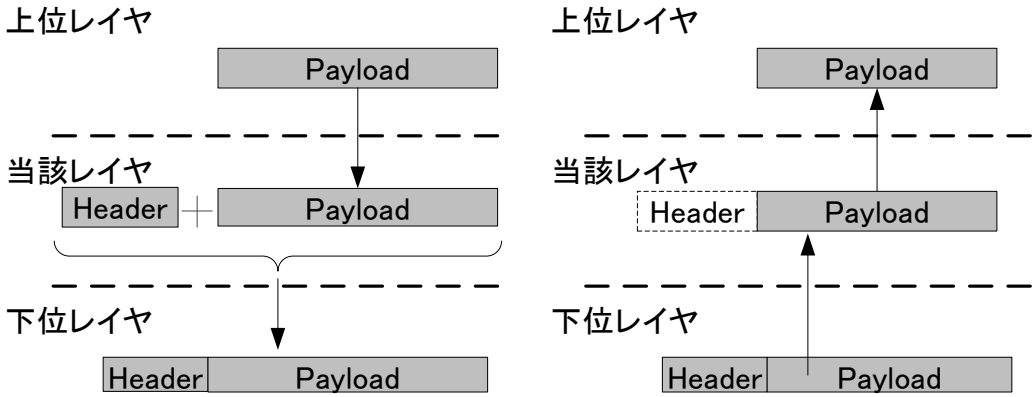

図 10　ヘッダの追加(左図)と削除(右図)

QualNet ではヘッダの追加処理を行う API として MESSAGE_AddHeader 関数が、ヘッダの削除処理を行う API として MESSAGE_RemoveHeader 関数が用意されている。これらのコードは 2.3 節で既に出てきたがここで再掲する。コードを見て分かるように、これらの関数は実際にはパケット先頭アドレスを指すポインタとパケットサイズをヘッダ分だけずらしているだけで、本当にヘッダ領域のメモリの確保と解放をその都度行っているわけではないことが分かる。そのため、例えば MESSAGE_RemoveHeader 関数を呼び出す前にパケット先頭アドレスを自身で用意した変数に保存しておけば、Header を剥がした後でもその値を参照することが実は可能である。

message.cpp

```
1394 void MESSAGE_AddHeader(Node *node,
1395                         Message *msg,
1396                         int hdrSize,
1397                         TraceProtocolType traceProtocol)
1398 {
1399     msg->packet -= hdrSize;
1400     msg->packetSize += hdrSize;
1401
1402     if (msg->isPacked)
1403     {
1404         msg->actualPktSize += hdrSize;
1405     }
1406     msg->headerProtocols[msg->numberOfHeaders] = traceProtocol;
1407     msg->headerSizes[msg->numberOfHeaders] = hdrSize;
1408     msg->numberOfHeaders++;
1409
1410     if (msg->packet < msg->payload) {
1411         char errorStr[MAX_STRING_LENGTH];
1412         sprintf(errorStr, "Not enough space for headers.\n"
1413                 "Increase the value of MSG_MAX_HDR_SIZE and try again.\n");
1414         ERROR_ReportError(errorStr);
1415     }
1416 }
```

message.cpp

```
1431 void MESSAGE_RemoveHeader(Node *node,
1432                           Message *msg,
1433                           int hdrSize,
1434                           TraceProtocolType traceProtocol)
1435 {
1436     ERROR_Assert(msg->headerProtocols[msg->numberOfHeaders-1] == traceProtocol,
1437                  "TRACE: Removing trace header that doesn't match!\n");
```

```
1438
1439     msg->packet += hdrSize;
1440     msg->packetSize -= hdrSize;
1441
1442     if (msg->isPacked)
1443     {
1444         msg->actualPktSize -= hdrSize;
1445     }
1446
1447     msg->numberOfHeaders--;
1448
1449     if (msg->packet > (msg->payload + msg->payloadSize)) {
1450         char errorStr[MAX_STRING_LENGTH];
1451         sprintf(errorStr, "Packet pointer going beyond allocated memory.\n");
1452         ERROR_ReportError(errorStr);
1453     }
1454 }
```

実例として IP 層で IP ヘッダを追加する箇所と IP ヘッダを削除する箇所のコードを示す。

以下は IP ヘッダを追加しているコードであるが、18930 行目で MESSAGE_AddHeader 関数を呼び出して、msg に hdrSize 分のヘッダ領域を追加している。その後 18932 行目以降で ipHeader というポインタ変数を使用して IP ヘッダ内の各フィールドに値を設定している様子が確認できる。

network_ip.cpp

```
18916 void
18917 AddIpHeader(
18918     Node *node,
18919     Message *msg,
18920     NodeAddress sourceAddress,
18921     NodeAddress destinationAddress,
18922     TosType priority,
18923     unsigned char protocol,
18924     unsigned ttl)
18925 {
18926     NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
18927     IpHeaderType *ipHeader;
18928     int hdrSize = sizeof(IpHeaderType);
18929
18930     MESSAGE_AddHeader(node, msg, hdrSize, TRACE_IP);
18931
18932     ipHeader = (IpHeaderType *) msg->packet;
18933     memset(ipHeader, 0, hdrSize);
18934
18935     IpHeaderSetVersion(&(ipHeader->ip_v_hl_tos_len), IPVERSION4) ;
18936     ipHeader->ip_id = ip->packetIdCounter;
18937     ip->packetIdCounter++;
18938     ipHeader->ip_src = sourceAddress;
18939     ipHeader->ip_dst = destinationAddress;
18940
18941 #ifdef ADDON_DB
18942     StatsDBAddMessageAddrInfo(node, msg, sourceAddress, destinationAddress);
18943 #endif // ADDON_DB
18944     if (ttl == 0)
18945     {
18946         ipHeader->ip_ttl = IPDEFTTL;
18947     }
18948     else
18949     {
18950         ipHeader->ip_ttl = (unsigned char) ttl;
18951     }
18952
18953     // TOS field (8 bit) in the IPV4 header
```

```
18954     IpHeaderSetTOS(&(ipHeader->ip_v_hl_tos_len), priority);
18955
          ... 中略 ...
19003     ipHeader->ip_p = protocol;
19004
19005     ERROR_Assert(MESSAGE_ReturnPacketSize(msg) <= IP_MAXPACKET,
19006                  "IP datagram (including header) exceeds IP_MAXPACKET
bytes");
19007
19008         IpHeaderSetIpLength(&(ipHeader->ip_v_hl_tos_len),
19009             MESSAGE_ReturnPacketSize(msg));
19010         unsigned int hdrSize_temp= hdrSize/4;
19011         IpHeaderSetHLen(&(ipHeader->ip_v_hl_tos_len), hdrSize_temp);
19012         //original code
19013         //SetIpHeaderSize(ipHeader, hdrSize);
19014
19015
19016 }
```

以下は、IP ヘッダを削除しているコードであるが、6944 行目で MESSAGE_RemoveHeader 関数を呼び出して、msg から IpHeaderSize(ipHeader)分のヘッダ領域を削除している様子が確認できる。

network_ip.cpp

```
6926 void
6927 NetworkIpRemoveIpHeader(
6928     Node *node,
6929     Message *msg,
6930     NodeAddress *sourceAddress,
6931     NodeAddress *destinationAddress,
6932     TosType *priority,
6933     unsigned char *protocol,
6934     unsigned *ttl)
6935 {
6936     IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;
6937
6938     *priority = IpHeaderGetTOS(ipHeader->ip_v_hl_tos_len);
6939     *ttl = ipHeader->ip_ttl;
6940     *protocol = ipHeader->ip_p;
6941     *sourceAddress = ipHeader->ip_src;
6942     *destinationAddress = ipHeader->ip_dst;
6943
6944     MESSAGE_RemoveHeader(node, msg, IpHeaderSize(ipHeader), TRACE_IP);
6945 }
```

2.4.3.4 ノード間での信号のやり取り

ここまでの処理は全て、同一ノード内のレイヤ間でのパケットのやり取りであったが、送信側ノードで PHY 層まで降りてきたパケットは、その下位に存在する Propagation レイヤに信号として伝えられ、Propagation レイヤで伝搬遅延を伴って、最終的に受信側ノードの PHY 層に信号として伝えられる。以下では、この送受信ノード間における信号のやり取りについて、イベントハンドリングの観点から解説する。なお、以下の説明は無線リンクを例に行っており、有線リンクを介したノード間の信号のやり取りの説明には必ずしもそのまま当てはまらないということを先に断わっておく。

まず例として、ある 1 つのチャネルを共有している 1 台の送信ノード（TX）と 2 台の受信ノード（RX1 および RX2）を考える。RX1 と RX2 はそれぞれ TX から距離 100m と 200m の位置にいるものと仮定する。このような状況において、TX の MAC 層が PHY 層にパケットを引き渡してから RX1 および RX2 の

MAC 層にパケットが届くまでの一連の処理を考えることにする。図 11 に、この状況における送受信ノード間でのイベント登録とイベント発生のイメージを示す。図の横方向は時間軸である。

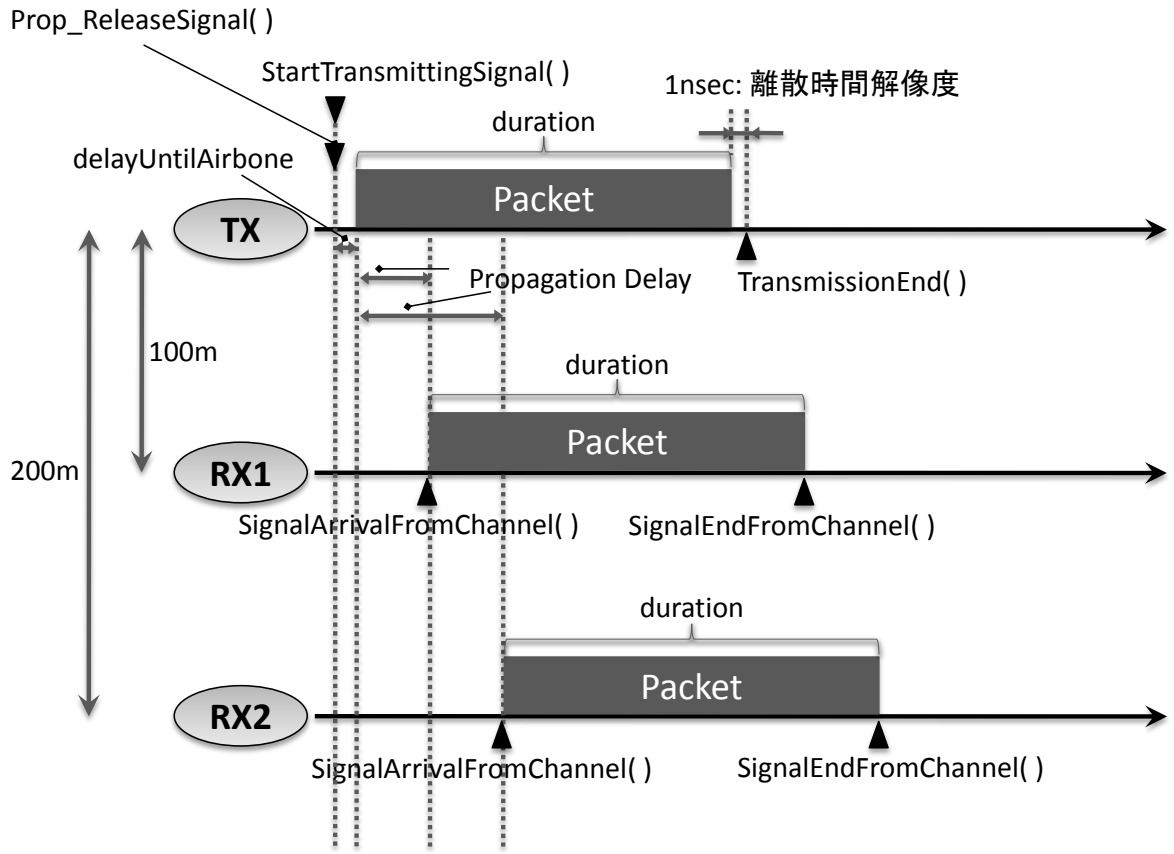

図 11　ノード間のパケット送受信に関わるイベント処理

【TX 側処理】

TX ノードの MAC 層は、PHY_StartTransmittingSignal 関数を呼び出して直接 PHY 層にパケットメッセージを引き渡す。PHY 層では対象 PHY プロトコルの StartTransmittingSignal 関数を呼び出す。以下では PHY 層プロトコルとして Abstract PHY を例にとって説明する。

phy_abstract.cpp

```
4038 void PhyAbstractStartTransmittingSignal(
4039     Node* node,
4040     int phyIndex,
4041     Message* packet,
4042     clocktype duration,
4043     BOOL useMacLayerSpecifiedDelay,
4044     clocktype initDelayUntilAirborne)
4045 {
         ... 中略 ...

4057     clocktype delayUntilAirborne = initDelayUntilAirborne;
4058     PhyData* thisPhy = node->phyData[phyIndex];
4059     PhyDataAbstract* phy_abstract = (PhyDataAbstract*)thisPhy->phyVar;
4060     PhyAbstractStats* stats = &(phy_abstract->stats);
4061     int channelIndex;
4062     Message *endMsg;

         ... 中略 ...

4152     PROP_ReleaseSignal(
```

```
4153          node,
4154          packet,
4155          phyIndex,
4156          channelIndex,
4157          phy_abstract->txPower_dBm,
4158          duration,
4159          delayUntilAirborne);

          ... 中略 ...

4171      endMsg = MESSAGE_Alloc(node,
4172                             PHY_LAYER,
4173                             0,
4174                             MSG_PHY_TransmissionEnd);
4175
4176      MESSAGE_SetInstanceId(endMsg, (short) phyIndex);
4177      MESSAGE_Send(node, endMsg, delayUntilAirborne + duration + 1);

          ... 中略 ...

4191 }
```

PhyAbstractStartTransmittingSignal 関数では、4152-4159 行目において、PROP_ReleaseSignal 関数を呼び出して無線チャネル(Propagation レイヤ) 上に信号を送信する処理を実行している。
PROP_ReleaseSignal 関数は、イベントスケジューラが提供している API 関数で、イベントスケジューラに信号送信イベントを通知する API である。引数の duration は信号を送信し続ける期間を表し、引数の delayUntilAirborne は PHY 層が信号を送信しようとした瞬間から本当に空気中に電波として送出されるまでの遅延時間(ケーブルやアンテナなどの送信装置媒体内の伝搬遅延など)を表している。イベントスケジューラは、これらの時間を加味して受信側ノードの PHY 層に後述する信号受信イベントを発生させる。

4171-4177 行目は、信号の送信完了を送信側ノード(つまり自身)に知らせるためのイベント登録処理である。ここで 4177 行目で呼び出している MESSAGE_Send 関数の引数を見ると分かるように、このイベントが発生する時刻は現在時刻から delayUntilAirborne と duration を足した時間ではなく、さらに 1nsec 加えた時間後となる。この+1nsec という値は QualNet の離散時間の解像度が 1nsec であることに起因している。つまり、信号を送信し続ける期間が明けた次の瞬間に当該イベントを発生させたいわけで、離散事象型シミュレータである QualNet において「次の瞬間」というは「次の離散時間解像度後」という扱いになる。間違えやすい点だが、duration は「信号の送信完了までの時間」ではなくあくまで「信号を送信し続ける時間」である点は理解しておく必要がある。

【RX 側処理】

TX ノードによる PROP_ReleaseSignal 関数を用いた信号送信イベントの通知を受けたイベントスケジューラは、信号受信対象となる各ノード(図の例では RX1 ノードおよび RX2 ノード)に信号受信開始イベント(PHY_SignalArrivalFromChannel 関数呼び出し)と信号受信終了イベント(PHY_SignalEndFromChannel 関数呼び出し)の 2 つのイベントをしかるべき時刻に発生させる。
各 RX ノードに対する PHY_SignalArrivalFromChannel 関数の呼び出しタイミングは TX ノードとの距離によって計算される PropagationDelay の時間後となり、PHY_SignalEndFromChannel 関数の呼び出しタイミングは PropagationDelay+duration の時間後となる。

まず PHY_SignalArrivalFromChannel 関数のコードを以下に示す。

この関数では、受信インタフェースの PHY モデル種別に応じた SignalArrivalFromChannel 関数を呼び出していることが分かる。

phy.cpp

```
2496 void PHY_SignalArrivalFromChannel(
2497    Node* node,
2498    int phyIndex,
2499    int channelIndex,
2500    PropRxInfo *propRxInfo)
2501 {
     ... 中略 ...

2524     node->enterInterface(node->phyData[phyIndex]->macInterfaceIndex);
2525
2526     switch(node->phyData[phyIndex]->phyModel) {
2527 #ifdef WIRELESS_LIB
2528         case PHY802_11b:
2529         case PHY802_11a: {
2530             Phy802_11SignalArrivalFromChannel(
2531                 node, phyIndex, channelIndex, propRxInfo);
2532
2533             break;
2534         }
2535         case PHY_ABSTRACT: {
2536 #ifdef ADDON_NGCNMS
2537             BOOL msgSendIgnore;
2538 #endif
2539             PhyAbstractSignalArrivalFromChannel(
2540                 node,
2541                 phyIndex,
2542                 channelIndex,
2543                 propRxInfo
2544 #ifdef ADDON_NGCNMS
2545                 , &msgSendIgnore
2546 #endif
2547                 );
2548
2549             break;
2550         }
2551 #endif // WIRELESS_LIB
     ... 中略 ...
2624         default: {
2625             ERROR_ReportError("Unknown or disabled PHY model\n");
2626         }
2627     }//switch//
2628     node->exitInterface();
2629 }
```

PHY モデル種別に応じた SignalArrivalFromChannel 関数の一例として、Abstract PHY の場合に呼び出される PhyAbstractSignalArrivalFromChannel 関数のコードを以下に示す。2074-2296 行目の case ブロック内の処理を見て分かる通り、この関数では受信電力／干渉電力の計算を行うだけでパケットメッセージの処理はまだ行われない。(引数としてもパケットメッセージは渡されていない。)

phy_abstract.cpp

```
1688 void PhyAbstractSignalArrivalFromChannel(
1689     Node* node,
1690     int phyIndex,
1691     int channelIndex,
1692     PropRxInfo *propRxInfo
1693 #ifdef ADDON_NGCNMS
1694     ,
1695     BOOL *msgSend
1696 #endif
```

```
1697     )
1698 {
         ... 中略 ...
1916     switch (phy_abstract->mode) {
             ... 中略 ...
2074         case PHY_IDLE:
2075         case PHY_SENSING:
2076         {
2077             double rxPowerInOmni_mW =
2078             NON_DB(ANTENNA_DefaultGainForThisSignal(node, phyIndex,
propRxInfo) +
2079                 propRxInfo->rxPower_dBm);
                 ... 中略 ...
2097             if (rxPowerInOmni_mW >= phy_abstract->rxSensitivity_mW) {
                     ... 中略 ...
2137                 PHY_SignalInterference(
2138                         node,
2139                         phyIndex,
2140                         channelIndex,
2141                         propRxInfo->txMsg,
2142                         &rxPower_mW,
2143                         &(phy_abstract->interferencePower_mW));
2144                 if (rxPower_mW >= phy_abstract->rxThreshold_mW) {

2151                 PhyAbstractLockSignal(
2152                     node,
2153                     phy_abstract,
2154 #ifdef ADDON_NGCNMS
2155                     propRxInfo,
2156 #endif
2157                     propRxInfo->txMsg,
2158                     rxPower_mW,
2159                     (propRxInfo->rxStartTime + propRxInfo->duration)
2160 #ifdef ADDON_NGCNMS
2161                     ,
2162                     channelIndex,
2163                     phyIndex
2164 #endif
2165
2166                     );

2225                 }
2226             }
2227             if (rxPower_mW < phy_abstract->rxThreshold_mW) {
                     ... 中略 ...
2253                 phy_abstract->interferencePower_mW += rxPower_mW;
2254                 stats->totalInterference_mW += rxPower_mW;
2255                 stats->numUpdates++;
                     ... 中略 ...
2292             }//if//
2293
2294
2295             break;
2296         }
2297
2298         default:
2299             abort();
2300
2301     }//switch (phy_abstract->mode)//
2302 }
```

次に PHY_SignalEndFromChannel 関数のコードを示す。

この関数では、PHY_SignalArrivalFromChannel 関数と同様に、受信インタフェースの PHY モデル種別に応じた SignalEndFromChannel 関数を呼び出していることが分かる。

phy.cpp

```
2632 void PHY_SignalEndFromChannel(
2633    Node* node,
2634    int phyIndex,
2635    int channelIndex,
2636    PropRxInfo *propRxInfo)
2637 {
     ... 中略 ...
2661     node->enterInterface(node->phyData[phyIndex]->macInterfaceIndex);
2662
2663     switch(node->phyData[phyIndex]->phyModel) {
2664 #ifdef WIRELESS_LIB
2665         case PHY802_11b:
2666         case PHY802_11a: {
2667             Phy802_11SignalEndFromChannel(
2668                 node, phyIndex, channelIndex, propRxInfo);
2669
2670             break;
2671         }
2672         case PHY_ABSTRACT: {
2673             PhyAbstractSignalEndFromChannel(
2674                 node, phyIndex, channelIndex, propRxInfo);
2675
2676             break;
2677         }
2678 #endif // WIRELESS_LIB
     ... 中略 ...
2749         default: {
2750             ERROR_ReportError("Unknown or disabled PHY model\n");
2751         }
2752     }//switch//
2753
2754     node->exitInterface();
2755 }
```

受信インタフェースの PHY モデル種別に応じた SignalEndFromChannel 関数の一例として、Abstract PHY の場合に呼び出される PhyAbstractSignalEndFromChannel 関数のコードを以下に示す。2682-2752 行目で受信パケットのエラー判定を行い、正常受信の場合（エラーが発生していない場合）に 3019-3074 行目で MAC_ReceivePacketFromPhy 関数を呼び出して MAC 層に受信パケットを引き渡している様子が確認できる。

phy_abstract.cpp

```
2481 void PhyAbstractSignalEndFromChannel(
2482     Node* node,
2483     int phyIndex,
2484     int channelIndex,
2485     PropRxInfo *propRxInfo)
2486 {
         ... 中略 ...
2682     if (phy_abstract->mode == PHY_RECEIVING) {
2683         if (phy_abstract->rxMsgError == FALSE) {
                 ... 中略 ...
2705                 phy_abstract->rxMsgError =
2706                     PhyAbstractCheckRxPacketError(
2707                         node,
2708                         phy_abstract,
2709
2710 #ifdef ADDON_BOEINGFCS
2711 #if MAC_ABSTRACT_ENABLED
2712                         &snr,
2713 #endif /* MAC_ABSTRACT_ENABLED */
2714 #endif /* ADDON_BOEINGFCS */
2715
```

```
2716                            &sinr);
                ... 中略 ...
2751            }//if
2752        }//if//
2753
2754        receiveErrorOccurred = phy_abstract->rxMsgError;
2755        //
2756        // If the phy is still receiving this signal, forward the frame
2757        // to the MAC layer.
2758        //
2759
2760        if ((phy_abstract->mode == PHY_RECEIVING) &&
2761                (phy_abstract->rxMsg == propRxInfo->txMsg))
2762        {
                ... 中略 ...
2994            PhyAbstractUnlockSignal(
2995    #ifdef ADDON_NGCNMS
2996                    node,
2997    #endif
2998                    phy_abstract);
                ... 中略 ...
3019            if (!receiveErrorOccurred) {
                    ... 中略 ...
3028                newMsg = MESSAGE_Duplicate(node, propRxInfo->txMsg);
                    ... 中略 ...
3048                MAC_ReceivePacketFromPhy(
3049                    node,
3050                    node->phyData[phyIndex]->macInterfaceIndex,
3051                    newMsg, phyIndex);
                    ... 中略 ...
3074            }
3075            else {
                    ... 中略 ...
3101            }
3102        }
3103        else {
                ... 中略 ...
3157        }//if//
3158    }
```