Design Document for RTIP/RMTP

Dinesh Verma
Hui Zhang

The Tenet Group
Computer Science Division
University of California, Berkeley
and
International Computer Science Institute, Berkeley

1. Introduction

1.1. Contents

This document describes the Real Time Internet Protocol (RTIP), Real Time Message Transport Protocol (RMTP) and Real Time Control and Management Protocol (RTCMP) suites for real-time communication. The protocols are designed to provide guaranteed performance communication in a packet-switched environment. All communication takes place on simplex connections called real-time channels (or simply channels).

Section 1 describes the environment and architecture under which the protocols operate. Section 2 presents the logical design of RMTP and Section 3 presents a logical design of RTIP. Section 4 discusses the state machines for RTIP and RMTP. Section 5 gives a detailed description of the states kept by RMTP and RTIP and their role in the state machines. Section 6 discusses the functionality of RTCMP in brief. Finally, section 7 provides some guidelines which should be followed in the implementation of these protocol suites.

1.2. Environment

The protocols work in association with Real Time Channel Administration Protocol (RCAP) which is responsible for channel admission and channel management. RMTP is a data transfer protocol which provides an unreliable guaranteed performance message service to the clients. RMTP functionalities are only present at the end-points (that is the source and the sink) of a real-time channel. RTIP provides a unreliable guaranteed performance packet service and its functionalities have to be present in all the sites involved in real-time communication. RTCMP is responsible for the management and control operations that are required while packets are being transferred on an established channel. RTCMP functionalities have to be present at all the sites in a network.

The protocols run in an environment which consists of a number of sites connected together by links or subnets. A site may be designated as either a host or a gateway. On a host, both RMTP and RTIP functionalities have to be supported, while gateways support only RTIP functionalities. Only hosts are allowed to be a source or a sink of any real-time channel. All the sites are identified using internet addressing scheme.

The protocol stack used for data communication is shown in Figure 1.

We assume that the performance guarantees provided by RMTP/RTIP are obeyed only when there are no hardware or software failures in any site or link.

We assume that the sites in a network can either go down or stay up, but do not undertake any malicious actions like transferring data with wrong channel identifiers. Any such malicious action would be considered equivalent to a software failure. We also assume that all the information stored at a site is lost

```
+++++++++++++++++++++++++                 ++++++++++++++++++
+    User Application   +                 +  Application   +
+                       +                 +                +
+++++++++++++++++++++++++                 ++++++++++++++++++
+       RMTP            +                 +     RMTP       +
+                       +                 +                +
+++++++++++++++++++++++++    +++++++++    ++++++++++++++++++
+       RTIP            +    + RTIP  +    +     RTIP       +
+                       +    +       +    +                +
+++++++++++++++++++++++++    +++++++++    ++++++++++++++++++
+       Physical net    +    + Net   +    +      Net       +
+                       ++++++       ++++++                +
+++++++++++++++++++++++++    +++++++++    ++++++++++++++++++

Host                            Gateway         Host
```

Figure 1 Protocol Stack

when it goes down, and can not be reconstructed when it comes up again. Thus, any failure of a site will stop the flow of data on all channels passing through that site. Any failure in a link or subnet is taken by a site as equivalent to the failure of all other sites sharing the same link or subnet.

We further assume that all the sites in the network have loosely synchronized clocks. The accuracy of clock synchronization is a lower bound on the delay jitter guarantees that can be supported by RTIP.

We further assume that the links connecting any two RTIP sites do not (or very infrequently) misorder packets. In those cases when the link is actually a subnet, a misordered packet is dropped by the receiving RTIP node. We assume that the probability of misordering packets is taken into consideration by RCAP at channel establishment time, and is acceptable to the client.

1.3. Relation Between Real-Time Protocols

All real-time data is carried on real-time channels. A channel state is kept in each node along its path. The RMTP state is maintained at the end points of the channel while the RTIP state is maintained at every node along the channel's path (including the end-points). The following functions regarding a channel's state are performed by each of the different protocol modules.

RCAP is responsible for initializing the state for a newly created channel and for destroying the state of a channel which is broken for any reason. RCAP calls routines (described in Section 5.2) to set-up the state for RTIP and RMTP, who do the actual initialization of the state.

RMTP maintains state at the end-points of the communication. RMTP state deals with regulation of the rate of packets entering the network, fragmentation and reassembly of messages into packets, and optional collection of end-to-end performance statistics about messages sent on a channel.

RTIP maintains state at every site along a channel's path. The RTIP state deals with verification of a channel's traffic rate, collection of per-node performance statistics and packet priority computations required for scheduling decisions.

RTCMP does not maintain any explicit per-channel state of its own, but monitors RTIP and RMTP states to ensure a smooth flow of data along established channels. RTCMP maintains state information about adjacent nodes to determine if there is any failure along the path of a channel. In some situations, RTCMP may request RCAP to delete an existing channel.

2. RMTP: Real Time Message Transport Protocol

2.1. RMTP service

RMTP provides a simplex end-to-end unreliable sequenced guaranteed performance message service. The incoming data channel consists of a number of messages which are delivered one message at a time to the receiver. RMTP is a component of RTTP (the Real Time Transport Protocol), the other part of RTTP deals with continuous media services and is called CMTP (see Figure 2).

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!                        !                               !
! CMTP                   !               RMTP            !
!                        !                               !
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!               RTIP                                     !
!                                                        !
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        Figure 2.
```

RMTP and CMTP present different interfaces to the RTTP user to access the guaranteed performance network provided by RTIP. This document only describes RMTP. CMTP is described in a companion document.

2.2. RMTP functionality

RMTP is responsible for end-to-end real-time communication functionality at both the sending and the receiving sides. These functionalities include message fragmentation and reassembly, optional collection of end-to-end performance statistics, and optional end-to-end checksums to verify the integrity of data being transmitted on the channel.

On the sending end, RMTP is responsible for:

1. Accepting messages from the sending application and fragmenting them into packets.
2. Inserting time-stamps in the message header in order for the receiving end to maintain performance statistics, if so desired.
3. Computing the checksum of the data to be sent out and putting it in the header, if so desired.
4. Providing sequence numbers to messages.
5. Regulating the flow of packets into RTIP so that they obey the traffic constraints promised at establishment time.

On the receiving end, RMTP is responsible for

1. Combining the packets together into messages
2. Collecting and maintaining message performance statistics about the channel, if so desired.
3. Verifying that the data checksums match, if applicable.
4. Determining the application which should receive a message and handing it over.

The following RMTP statistics are kept at the receiving side:

The number of messages received.
The number of messages lost.
The number of messages received within the desired delay bound.
The number of messages received within the desired delay jitter bound.

2.3. RMTP interfaces

The interfaces defined here are logical. The actual implementation of the RMTP layer may decide to provide these interfaces in a manner deemed most suitable for the particular environment. On the sending side, the client accesses RMTP by calling the primitive `rttp_send (lcid, data, len)`

where `lcid` is a number assigned to a channel which is unique locally on the sending host; `len` is the number of bytes in the current message and can not exceed the maximum message size determined for that channel at the establishment time; and `data` contains the actual data to be included in the message.

On a `rttp_send`, RMTP attempts to schedule the packet(s) generated by this message to be handed over to RTIP. On the successful scheduling of the packets in the message, `rttp_send` returns the number of bytes successfully scheduled. Otherwise `rttp_send` signals an error if one of the following cases arise:

:the message was too long and therefore was not sent.
:the message could not be scheduled because of lack of buffer space.
:an attempt was made to send on a `lcid` which does not exist or is not owned by the sender.

On the receiving side, the application can receive the data by calling a primitive:

`rttp_receive(lcid, data, len)`

where `lcid` is a number assigned to a channel which is unique locally on the receiving host; `data` is a pointer to a buffer which has at least `len` bytes of space. RMTP copies as much of the message which will fit in the buffer. To ensure that the entire message is received, the client is recommended to always provide a buffer large enough to hold the largest message possible on the channel. (this value is determined at channel establishment time).

Each `rttp_receive` returns exactly one message to the application. An error flag is returned when a message is found to be missing. `rttp_receive` may signal errors in the following cases:

:an attempt was made to receive on a `lcid` which does not exist or is not owned by the receiver
:the parameter `len` did nor provide enough space for the message and only part of it was received. The rest of data is dropped.
:the next message on the channel was missing.

The delay of a message is defined as the difference between the absolute time when the packet is available to be received by the RMTP receiving client, and the time when the send was called. The notion of delay is not defined for messages that are dropped for any reason. The delay of a message is measured by means of a time-stamp put in the message header by the sending client. Lost messages are determined by checking if there are any gaps in the sequence numbers of received messages.

Two options can be specified by a RMTP client at connection establishment time. These options are kept in the RMTP state at the source and the sink of a channel. The options can not be changed when data is being transferred on the connection. The options are:

`DATA_CHECK_SUM`: when this option is turned on, RMTP on the sending side places the checksum for the entire message (i.e. RMTP header and user data) into the RMTP header. When the option is turned off, the checksum field in RMTP header covers only the RMTP header itself. (See Section 2.4 for RMTP header fields). The default value of this option is off.

`STATISTICS_COLLECTION`: when this option is turned on, RMTP on the sending side places a time-stamp in the RMTP header every time a message is received. RMTP on the receiving side uses the time-stamp to determine the end-to-end delay experienced by this message. RMTP at the receiving end also updates the various statistics kept in its state on the receipt of every packet. The default value of this option is off.

In order to expedite the fragmentation and reassembly of packets, RMTP expects the message sequence number and an `end_of_message` flag to be carried in each packet. One word of information is passed to RTIP by RMTP. This word has the information that is needed on a per-packet basis by RMTP. The first byte representing the message sequence number and the first bit in the second byte representing the `end_of_message` flag. The other two bytes are unused at this moment. One possible use for these bytes in the future would be to carry a per-packet data checksum which can be used to deliver partially correct messages to the RMTP client.

2.4. Header format

RMTP has the header format shown in Figure 3.

```
0               7                15               23              31
!+++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
+                        Time Stamp                               !
++++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
+       Message length           !Mess Seq #      ! Options       !
++++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
+               Checksum                                          !
++++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
+               Variable length user data                         !
++++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
```

Figure 3

The fields in RMTP header serve the following functions:

`Time Stamp` is put into a RMTP message if the channel has the `STATISTICS_COLLECTION` option turned on. The time-stamp is a 32 bit number which indicates the absolute time when the `rttp_send` primitive was called by the client at the sending end. The granularity of the time-stamp is 2^{-16} seconds. Thus, the maximum time-stamp value can be 18 hours. This implies that no message should be delayed in the network for more than 18 hours by RMTP/RTIP.

`Message Length` gives the length of a message in terms of the number of packets it has been fragmented into. A packet can be fragmented into at most 2^{16} packets, and each packet can be at most 2^{16} bytes long. Thus, the maximum RMTP message can be 2^{32} bytes or 4.3 Giga-bytes long.

`Message Sequence #` is a 8 bit sequence number which can be used to detect a lost message.

`Options` field is currently unused by RMTP, but can be used to pass control information in data messages.

`Checksum` is a CRC computed over the header of the message. If the `DATA_CHECK_SUM` option is turned on, the CRC is done over the entire length of the message, including the RMTP header. For efficient hardware implementation, the checksum should be part of the trailer, when used for checksumming data. However, we have placed it in the header for the first version of the protocol.

RMTP also needs some information to be carried in every packet (fragment) of a message. The format of this information is shown in Figure 3a.

```
0               7                15               23              31
!+++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
! Message Seq # ! Flags          !         reserved               !
!+++++++++++++++!++++++++++++++++!++++++++++++++++!+++++++++++++++!
```

Figure 3a

The following fields are used in the per-packet information:

`Message Seq #`: determines which message a packet belongs to.

`Flags`: Two bits in the Flags are used to indicate the beginning and the end of a message. Bit number 0 is the `beginning_of_message` flag while bit number 1 is the `end_of_message` flag. The other bits are reserved for future use.

The other two bytes are reserved for future use.

3. RTIP: Real Time Internet Protocol

3.1. RTIP services

RTIP provides sequenced end-to-end unreliable guaranteed performance packet service. The incoming data channel consists of a number of packets which are delivered one packet at a time to the receiver by an indication function registered at channel establishment time.

3.2. RTIP functionality

At each node, RTIP performs the following functions:

1. Verifying that the packets arrive into the RTIP node according to the traffic contract signed at channel establishment time.
2. Holding packets till they become eligible for transmission
3. Scheduling eligible packets according to their priority.
4. Transfer the packet to the next node or the upper layer (at the last RTIP node)
5. Optional collection of per-node packet statistics, and end-to-end packet statistics at the destination host.

3.3. RTIP interface

RTIP exports the following interface to higher layer protocols.

The send primitive has the form:

```
rtip_send(lcid,pkt, len, eligib_time,higher_level_encaps)
```

where `lcid` is the local channel identifier for the channel, `pkt` is the actual data to be transferred and `len` is its length. `higher_level_encaps` can contain some information which needs to be carried on a per-packet basis from the upper layer. The RTIP header format restricts this argument to be at most 2 bytes long. An example of a higher level field is the message sequence number and/or the `end_of_message` flag which needs to be known to RMTP on a per-packet basis. The 2 bytes of higher level encapsulation can be used in any way by the higher level protocol (RMTP or CMTP) in any fashion.

The parameter of `eligib_time` needs some explanation. Ideally, RMTP should space out the packets created by the fragmentation process according to the traffic parameters specified at connection establishment time. However, actual physical separation of the packets may be difficult to achieve in a system. Therefore, we allow the RMTP module to send packets with a tag that describes the actual time that the packet was supposed to be handed over to RTIP by RMTP. This actual time when the packet was to be handed over to RTIP is specified by means of the `eligib_time` parameter.

On the receipt of a packet, an `indication` is given to the upper layer

```
rtip_indication(lcid,pkt,len,pkt_seq_no,higher_level_encaps)
```

where `lcid` is the local channel identifier for the channel, `pkt` is the actual data received and `len` is its length. The higher level field entries in the packet and packet sequence number are passed on to the higher layer as well.

The delay of a RTIP packet is defined as the time difference between the `eligib_time` used in the send operation and the time instant the `rtip_indication` routine was called by the RTIP module at the receiver. For packets lost in the network, `rtip_indication` is never called and their delays are undefined.

The options field allows RMTP (or any other higher layer) to send up to 2 bytes of information that can be carried per packet. For RMTP, this information is the message sequence number and bits marking end of messages.

3.4. Header format

The header format for RTIP is shown in Figure 4.

The fields in the header are as follows

```
0       3       7               15              23              31
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
!RTIP   ! Ver   ! unused        ! Local id                      !
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
! Packet Length                 ! Pkt Seq#                      !
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
!                       Time Stamp                              !
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
!                            Higher Layer fields                !
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
!         reserved              ! Header Checksum               !
!+++++++++++++++!+++++++++++++++!+++++++++++++++!+++++++++++++++!
```

Figure 4.

`RTIP` is a 4 bit field which is used by the interfaces in gateways and hosts to indicate that the packet is to be delivered to the RTIP module.

`Ver` is the RTIP version number. The current version is Version 0.

Bits 8 to 16 are unused in this version.

`Local id` is the local channel identifier at the next downstream site along the path of the channel. It is a 16 bit number which must be determined by RCAP at channel establishment time. The local channel identifier uniquely identifies the channel for the next RTIP site on the path. This value must be updated at each of the site along the path of a channel.

`Packet Length` is a 16 bit number denoting the length of the packet in bytes. The maximum packet size is 2^{16} bytes or 64 KB.

`Pkt Seq#` is a 16 bit number which is monotonically increasing by one between consecutive packets on the same channel. This can be used to detect any lost packets on a channel.

`Time Stamp` is a 32 bit number that carries the time the packet was received by the RTIP module at the sending host. The time-stamp has a granularity of 2^{-16} seconds.

`Higher Layer fields` allows up to 4 bytes of any higher level protocol information to be carried in every RTIP packet. For RMTP, the first byte contains the sequence number of the message to which this packet belongs, while the next byte carries flags (for example the `end_of_message` flag). RTIP does not interpret the entries in this field.

`Header Checksum` is a 16 bit CRC computed over all the fields in the RTIP header.

4. State Machines of RTIP and RTTP

There are three RTIP machines: rtip_RECEIVING_MACHINE, rtip_RATE_MACHINE and rtip_TRANSMISSION_MACHINE. Rtip_RECEIVING MACHINE and rtip_RATE_MACHINE communicate by operating on the same calendar queue. Rtip_TRANSMISSION_MACHINE is actually the driver machine, it needs to keep three transmission queues.

1. deterministic queue
2. statistical queue
3. non-real-time queue

Rtip_RECEIVING_MACHINE and Rtip_RATE_MACHINE are the producers of the transmission queues, Rtip_TRANSMISSION_MACHINE is the consumer of the transmission queues.

There are three RTTP machines: rttp_SENDING_MACHINE, rttp_LEAKY_MACHINE and rttp_RECEIVING_MACHINE. Rttp_SENDING_MACHINE and rttp_LEAKY_MACHINE are the producer and consumer of the receiving queue respectively. On the sending side, rttp_RATE_MACHINE passes packets to rtip_RECEIVING_MACHINE; on the receiving side, rttp_RECEIVING_MACHINE gets packets from rtip_RECEIVING_MACHINE and rtip_RATE_MACHINE.

The relationships between all the six machines are shown in Figure 5.

```
    !-----------!                                        !-----------!
    !           !                                        !           !
!---!rttp_SendM !                            !---------->!rttp_RevM  !
!   !           !                            !           !           !
!   !-----------!                            !           !-----------!
!                                            !
!      --------------                        !
!-->    Leaky bucket  --!                    !
       --------------   !                    !
                        !                    !
!----------------------!                     !
!                                            !
!                                            !
!   !-----------!                            !
!   !           !                            !
!-->!rttp_LeakM !-------!                    !
    !           !       !                    !
    !-----------!       !                    !
                        !                    !
                        !                    !
!----------------------!                     !
!                                            !
!                                            !
!                                            !
!   !-----------!                            !
!-->!           !                            !
    !rtip_RevM  !----------------------------!
!---!           !
!   !-----------!
!
!      --------------                                    !-----------!
!-->   Calendar Queue --!        ---------------         !           !
       --------------   !  !--->  Transmission Queues -> !rtip_TrM   !
!----------------------!   !      ------------------     !           !
!                          !                             !-----------!
!                          !
!   !-----------!          !
!   !           !          !
!-->!rtip_RateM !----------!
    !           !
    !-----------!
```

Figure 5. Relations between Different Machines

4.1. rtip_RECEIVING_MACHINE

State Diagram:

```
   !--------!       t1           !--------!
   !        !------------------->!        !
---!   s0   !                    !   s1   !----
!  !        !<-------------------!        !   !
!  !------- !       t3           !--------!   !
!      ^ !                          !  ^      !
! t0   ! !                          !  !  t4  !
!------! !                          !  !------!
         !                          !
         !         !--------!       !
         !    t2   !        !   t5  !
         !-------->!   s2   !<------!
                ---!        !
                !  !------- !
                !      ^
                !  t6  !
                !------!
```

Events: e0) arrival of a packet
e1) return of a buffer by transmission machine
e2) detection of failure

States: s0) normal state with more than one free buffers
s1) normal state with no free buffers
s2) panic state

State Variables:
free_buf: number of free buffers available

Transitions:

t0) From s0 to s0
Condition: e0 AND (free_buf > 1)
Actions: 1. free_buf --;
2. rtip_rm_proc();

t1) From s0 to s1
Condition: e0 AND (free_buf == 1)
Actions: 1. free_buf --;
2. rtip_rm_proc();

t2) From s0 to s2
Condition: e2
Actions: call notify_error of RTCMP

t3) From s1 to s0
Condition: e1
Actions: 1. free_buf ++;

```
        t4) From s1 to s1
            Condition:          e0
            Actions:            1. overwrite buffer (or drop packet)

        t5) From s1 to s2
            Condition:          e2
            Actions:            call notify_error of RTCMP

        t6) From s2 to s2
            Condition:          e0
            Actions:            drop the packet

Rtip_rm_proc() {
        1. read the clock;
        2. if (packet sequence number < expected sequence number) {
                drop the packet;
                update statistics;
                return();
            } else if (packet sequence number > expected number) {
            /* a packet is missing, no action */
            }
        3. calculate the deadline and the holding time
        4. update the state:
                a) expected arrival time of next packet
                b) statistics
                c) sequence number
        5. update the header:
                a) next node LCID
        6. insert into calendar queue (not destination)
           or hand it to a higher layer protocol (destination)
} /* rtip_rm_proc */
```

4.2. rtip_RATE_MACHINE

```
State Diagram:



                      !--------!
                      !        !
                      !   s0   !----
                      !        !   !
                      !--------!   !
                           ^       !
                           !  t0   !
                           !-------!

Events: e0) timer goes off (a clock tick of the calendar queue)

States: s0) normal state

Transitions:
        t0) from s0 to s0
```

```
            Condition:          e0
            Actions:            1. rtip_rate_proc();

Rtip_rate_proc() {
       insert packet into corresponding transmission queue
} /* rtip_rate_proc */
```

4.3. rtip_TRANSMISSION_MACHINE

State Diagram:

```
    !--------!       t0             !--------!
    !        !---------------------->!        !
    !   s0   !<----------------------!   s1   !
    !        !       t1        -----!        !----!
    !--------!                 !    !--------!    !
                               !       ^  ^       !
                               !  t3   !  !  t2   !
                               !-------!  !-------!
```

```
Events: e0) arrival of a packet
        e1) finishing transmission of a packet

States: s0) Idle state
        s1) Busy State

State Variables:
        packets_in_queue:       number of packets in all
                                three queues (not including
                                the one being serviced)

Transitions:
        t0) from s0 to s1
            Condition:          e0
            Actions:            1. packets_in_queue = 0;
                                2. rtip_tm_proc();


        t1) from s1 to s0
            Condition:          e1 AND (packets_in_queue == 0)
            Actions:

        t2) from s1 to s1
            Condition:          e0
            Actions:            1. packets_in_queue ++;

        t3) from s1 to s1
            Condition:          e1 AND (packets_in_queue > 0)
            Actions:            1. packets_in_queue --;
                                2. rtip_tm_proc();
```

```
Rtip_tm_proc() {

        1. if (deterministic queue) {
                transmit the first packet in deterministic queue
          } else if (statistical queue) {
                transmit the first packet in statistical queue
          } else
                transmit the first packet in non-real-time queue;
          }
        2. free the buffer

} /* rtip_tm_proc */
```

4.4. rttp_SENDING_MACHINE

State diagram

```
          !--------!          t2            !--------!
          !        !----------------------->!        !
          !   s0   !<-----------------------!   s1   !
     -----!        !----!     t4            !        !----!
     !    !--------!    !                   !--------!    !
     !      ^ ! ^       !                       ! ^       !
     !  t1  ! ! !  t0   !                       ! !  t5   !
     !------! ! !-------!                       ! !-------!
              !              !--------!         !
              !       t3     !        !   t6    !
              !------------->!   s2   !<--------!
                          ---!        !
                          !  !------- !
                          !      ^
                          !  t7  !
                          !------!
```

```
Events: e0) arrival of a message
        e1) return of a buffer by leaky bucket machine
        e2) failure notification by RTCMP

States: s0) normal state with no message waiting for free buffer
        s1) normal state with one message waiting for free buffer
        s2) panic state

State Variables:
        buf_size                size of free buffer

Transitions:
        t0) from s0 to s0
            Condition:          e0 AND (msg_size <= buf_size)
            Actions:            1. buf_size -= msg_size;
                                2. rttp_sm_proc();
```

```
        t1) from s0 to s0
            Condition:          e1
            Actions:            1. buf_size += return_buf_size;

        t2) from s0 to s1
            Condition:          e0 AND (msg_size > buf_size)
            Actions:            1. block the process

        t3) from s0 to s2
            Condition:          e2
            Actions:            call RTCMP "notify_error"

        t4) from s1 to s0
            Condition:          e1 AND (msg_size <=
                                        buf_size+return_buf_size)
            Actions:            1. buf_size += return_buf_size
                                                - msg_size;
                                2. rttp_sm_proc();
        t5) from s1 to s1
            Condition:          e1 AND (msg_size >
                                        buf_size+return_buf_size)
            Actions:            1. buf_size += return_buf_size;

        t6) from s1 to s2
            Condition:          e2
            Actions:            call RTCMP "notify_error"

        t7) from s2 to s2
            Condition:          e0
            Actions:            1. drop the packet

Rttp_sm_proc() {
        1. read clock
        2. fragment the message into packets
        3. allocate and fill in RTTP header
        4. update states
                a) message number
        5. insert the packets into leaky bucket
}
```

4.5. rttp_LEAKY_MACHINE

```
State Diagram

                       !--------!
                       !        !
                       !   s0   !----
                       !        !   !
                       !--------!   !
                            ^       !
                            !  t0   !
                            !-------!
```

```
    Events: e0) timer goes off

    States: s0) normal state

    Transitions:
        t0) from s0 to s0
            Condition:          e0
            Actions:            1. rttp_lb_proc();

    Rttp_lb_proc() {
        1. hand it to RTIP RECEIVING MACHINE
        2. free buffer to RTTP SENDING MACHINE
}
```

4.6. rttp_RECEIVING_MACHINE

The receiving machine presented here assumes that one message has just one packets, i.e., there is no reassembly.

```
State Diagram:



       !--------!       t5           !--------!
       !        !------------------->!        !
    ---!   s0   !                    !   s1   !----
t0  !  !        !<-------------------!        !   !
t1  !  !------- !       t7           !--------!   ! t8
t2  !       ^ !                         !  ^      ! t9
t3  !       ! !                         !  !      !
t4  !------! !                          !  !------!
             !                          !
             !         !--------!       !
             !    t6   !        !   t1  !
             !-------->!   s2   !<------!
                    ---!        !
                    !  !------- !
                    !      ^
                t11 !      !
                    !------!
```

```
Events: e0) arrival of a message
        e1) user receive a message
        e2) detection of fatal error

States: s0) normal state with more than one free buffers
        s1) normal state with no free buffers
        s2) panic state

State Variables:
        free_buf:       number of free buffers available
        u_msg_seq:      the sequence number of the message that is
                        to be returned upon the next receive
                        by the user
        r_msg_seq:      the sequence number of the next message
                        that is to be received from RTIP
        msg_rev[]:      an array indexed by message sequence number
                        The value of an element in the array can be:
                            REV:    the message has been received
                            NO_REV: the message has not been
                                    received
                            DROP:   the message has been dropped
                                    All the elements in the array
                                    are initialized to have the
                                    value NO_REV.
Transitions:
        t0) From s0 to s0
            Condition:          e0 AND (free_buf > 1) AND
                                   (checksum right OR no checksum)
            Actions:            1. free_buf --;
                                2. rttp_rm_proc();

        t1) from s0 to s0       e0 AND (checksum wrong)
                                1. drop the packet
                                2. for (i = r_msg_seq;
                                    i <= seq no of the msg; i++)
                                     msg_rev[i] = DROP;
                                3. if (seq no of the msg>=r_msg_no)
                                     r_msg_no = seq no of the msg+1;
        t2) From s0 to s0
            Condition:          e1 AND (msg_rev[u_msg_no] == DROP)
            Actions:            1. return an error message
                                2. u_msg_no ++;

        t3) From s0 to s0
            Conditions:         e1 AND (msg_rev[u_msg_no] == REV)
            Actions:            1. copy first message in queue
                                   to user space;
                                2. unblock the process
                                3. u_msg_no ++;
                                4. free_buf ++;

        t4) From s0 to s0
            Conditions:         e1 AND (msg_rev[u_msg_no] == NO_REV)
```

```
                        1. block the user process

t5) From s0 to s1
    Condition:          e0 AND (free_buf == 1)
                           AND (checksum right OR no_checksum)
    Actions:            1. free_buf --;
                        2. rttp_rm_proc();

t6) From s0 to s2
    Condition:          e2
    Actions:            call notify_error of RTCMP

t7) From s1 to s0
    Condition:          e1 AND (msg_rev[u_msg_no] == REV)
    Actions:            1.copy first message in queue
                          to user space;
                        2. unblock the process
                        3. u_msg_no ++;

t8) From s1 to s1
    Condition:          e1 AND (msg_rev[u_msg_no] == DROP)
    Actions:            1. return an error message
                        2. u_msg_no ++;
                        4. free_buf ++;

t9) From s1 to s1
    Condition:          e0
    Actions:            1. overwrite or drop packet


t10)From s1 to s2       e2
                        call notify_error of RTCMP

t11)From s2 to s2       e1
    Condition:          1. return error message
```

```
Rttp_rm_proc() {
        1. read the clock;
        2. if (r_msg_seq > sequence number of the packet) {
                drop the message;
                return();
           }
        3. if (r_msg_seq < sequence number of the packet) {
                for (i = r_msg_seq; i < seq no of the msg; i++)
                    msg_rev[i] = DROP;
                r_msg_seq = sequence number of the message;
           }
        4. msg_rev[r_msg_seq] = REV;
           r_msg_seq ++;
           queue the message at user/kernel interface (may trigger
           a receive action the client is waiting for data)
```

5. RTTP/RTIP States

The states of RTIP and RTTP are setup by RCAP at the channel establishment time. In this section, the pseudo code representations of RTIP and RTTP states are presented first before the relations between these states and the state machines presented in Section 4 are described.

5.1. State structure:

The channel state for RTIP consists of a structure:

```
typedef struct {
        trafficSpec trafficSpec;
        perfSpec perfSpec;
        channelID globalID;
        forwardingInfo forwardingInfo;
        rtipStr RTIP;
        void *transportState;
        u_char transportType;
        u_char  rtip_Options;
        bufferStr bufferStr; } rtipState;
```

The channel state for RTTP consists of a structure:

```
typedef struct {
        trafficSpec trafficSpec;
        perfSpec perfSpec;      /* end to end performance specs */
        channelID globalID;
        rttpStr RTTP;
        rtipState RTIPState;
        u_char rttp_Options;
        bufferStr bufferStr; } rttpState;
```

The traffic structure looks like:

```
typedef struct {
        u_int xmin;
        u_int xave;
        u_int I;
        u_int Smax; } trafficSpec;
```

The performance specs look like:

```
typedef struct {
        u_int d;                    /* local delay bound (node+link) */
        u_int dn;                   /* local delay bound (node only) */
        u_int adn;                  /* accumul total delay */
        u_int j;                    /* local jitter bound */
        u_int z;                    /* statistical delay probability */
        u_int w;                    /* statistical drop probability */
        u_int u;                    /* statistical jitter probability */
        u_char type;                /* channel type */ } perfSpec;
```

The global identifier is represented as

```
typedef struct {
        struct in_ddr src_ip;      /* source IP address */
        u_short src_cid;           /* source channel ID */ } channelID;
```

The forwarding information consists of:

```
typedef struct {
        u_short f_lcid;             /* downstream local channel ID */
        u_short f_ifn;              /* downstream net interface number*/
        struct ifnet *f_ifp;        /* downstream network interface */
        struct in_addr *f_ip;       /* downstream node IP address */
        u_short b_lcid;             /* upstream local channel ID */
        u_short f_ifn;              /* upstream net interface number */
        struct ifnet *b_ifp;        /* upstream network interface */
        struct in_addr *b_ip;       /* upstream node IP address */ }
forwardingInfo;
```

The RTIP-specific structure consists of the following structure

```
typedef struct {
        runtimeState run_state;
        perfMon monitor_state;
        perfMon end_to_end_state; } rtipStr;

typedef struct {
        u_int num_received;
        u_int num_dropped;
        u_int num_delayed;
        u_int num_early; } perfMon;

typedef struct {
        u_long  sum;
        u_int  history[];
        u_int  exp_arrival;
        u_short exp_seq; } runtimeState;
```

The RTIP options are following

```
DEBUG_FLAG      /* Collect per node Statistics */
DEBUG_END__FLAG /* Pkt Statistics at Recv.*/
DESTINATION     /* If this node is the last one on path */
```

```
typedef struct {
        u_int num_bufs;
        buf* buf_list; } bufferStr;
```

RTTP maintains the following structure at the end_points:

```
typedef {
        u_int msg_no;     /* Sequence number sent/ received  */
        u_int msg_no2;    /* seq no used by receiving client so far */
        char* msg_rev[];  /* Array of message buffers at the receiver */
        u_int pkt_no;     /* Expected sequence number of next packet */
```

```
        perfMon mesg_stats;

        struct protcl_sw upper_layer; }       rttpStr;
```

The RMTP options are defined as follows:

```
CKSUM_FLAG      /* Collect Packet Statistics */
MSG_DEBUG_FLAG  /* Collect Message Statistics */
CM_MODE_FLAG    /*  use CM interface      */
```

5.2. State Creation

The channel state is initialized by RCAP and passed to RTIP and RMTP by the procedures:

```
RTIPEstablish(rtip_state)
rtipState rtip_state;
```

and

```
RTTPEstablish(rmtp_state)
rttpState rmtp_state;
```

All the entries of the state, except the field RTIP are filled in by the RCAP module. RTIP will be allocated and initialized by a RTIP procedure initialize_rtip(state). The following entries are initialized by this procedure:

1. Buffers are allocated depending on the number passed by RCAP.
2. The performance structures are initialized to all zeros.
3. The runtime rate-control structures are initialized to all zeros.
4. A link between the rtipState and rttpState is made.

The channel state is deleted by the calls:

```
int RTIPReleaseResources(lcid)
u_short lcid;

int RTTPReleaseResources(lcid)
u_short lcid;
```

5.3. State structures and State Machines

The six state machines described in Section 4 are high level descriptions of the operations of RTTP and RTIP protocols. General terms referring to states are used in the state machines. In this section, we will setup the correspondence between the general terms used in state machines and the exact entries in the state structures.

5.3.1. rtip_RECEIVING_MACHINE

```
State Variables:
        free_buf    number of free buffers available
Rtip_rm_proc() {
        1. read the clock;
        2. if (packet sequence number < expected sequence number) {
                drop the packet;
                update statistics;
                return();
            } else if (packet sequence number > expected number) {
            /* a packet is missing, no action */
            }
```

```
        3. calculate the deadline and the holding time using
           expected arrival time and delay bound
        4. update the state:
                a) expected arrival time of next packet
                b) statistics
                c) sequence number
        5. update the header:
                a) next node LCID
        6. insert into calendar queue (jitter packet)
           or insert into a transmission queue
                (delay packet to be forwarded)
           or hand it to a higher layer protocol
                (delay packet at destination)
} /* rtip_rm_proc */
```

```
Correspondence:

    free_buf                        rtipState->BufferStr
                                    ->num_bufs
    expected sequence number        rtipState->RTIP
                                    ->run_state->history
    statistics                      rtipState->RTIP
                                    ->monitor_state
    delay bound                     rtipState->perfSpec
                                    ->d
    expected arrival time           rtipState->RTIP
                                    ->run_state->exp_arrival
    next node LCID                  rtipState->forwardingInfo
                                    ->f_licd
    channel type                    rtipState->perfSpec->type
```

5.3.2. rtip_RATE_MACHINE

```
    Rtip_rate_proc() {
            if (at destination)
                    hand it to a higher layer protocol
            else
                    insert  packet  into  deterministic  transmission
queue
    } /* rtip_rate_proc */

   Correspondence:
           at destination               rtipState->rtip_Options
```

5.3.3. rtip_TRANSMISSION_MACHINE

```
State Variables:
        packets_in_queue:       number of packets in all
                                three queues (not including
                                the one being serviced)
Rtip_tm_proc() {
```

```
        1. if (deterministic queue) {
                 transmit the first packet in deterministic queue
           } else if (statistical queue) {
                 transmit the first packet in statistical queue
           } else
                 transmit the first packet in best-effort queue;
           }
        2. free the buffer

} /* rtip_tm_proc */
```

Notice that rtip_TRANSMISSION_MACHINE is per output link instead of per channel, so the information in the channel states are not to be used.

5.3.4. rttp_SENDING_MACHINE

```
State Variables:
        buf_size                size of free buffer

Rttp_sm_proc() {
        1. read clock
        2. fragment the message into packets
        3. allocate RTTP and RTIP headers for each packet
        4. fill in RTTP header and partial RTIP header which
           includes the RTIP sequence number
        5. update states
                a) message number
                b) packet sequence number
        6. insert the packets into leaky bucket
}
```

```
Correspondence:

     buf_size                rttpState->bufferStr->num_bufs
     fragmentation info      rttpState->rtipState->trafficSpec
                             ->Smax
     message number          rttpState->RTTP->msg_no
     packet number           rttpState->rttpStr->pkt_no
```

5.3.5. rttp_LEAKY_MACHINE

```
Rttp_lb_proc() {
        1. hand it to RTIP RECEIVING MACHINE
        2. free buffer to RTTP SENDING MACHINE
}
```

5.3.6.

```
State Variables:
        free_buf:       number of free buffers available
        u_msg_seq:      the sequence number of the message that is
                        to be returned upon the next receive
                        by the user
        r_msg_seq:      the sequence number of the next message
                        that is to be received from RTIP
        msg_rev[]:      an array indexed by message sequence number
                        The value of an element in the array can be:
                            REV:    the message has been received
                            NO_REV: the message has not been
                                    received
                            DROP:   the message has been dropped
                                    All the elements in the array
                                    are initialized to have the
                                    value NO_REV.

    Rttp_rm_proc() {

        1. read the clock;
        2. if (r_msg_seq > sequence number of the packet) {
                drop the message;
                return();
           }
        3. if (r_msg_seq < sequence number of the packet) {
                for (i = r_msg_seq; i < seq no of the msg; i++)
                    msg_rev[i] = DROP;
                r_msg_seq = sequence number of the message;
           }
        4. msg_rev[r_msg_seq] = REV;
           r_msg_seq ++;
           queue the message at user/kernel interface (may trigger
           a receive action the client is waiting for data)


    Correspondence:
            free_buf                rttpState->bufferStr->num_bufs
            r_msg_seq               rttpState->RTTP->msg_no
            u_msg_seq               rttpState->RTTP->msg_no2
            msg_rev[]               rttpState->RTTP->msg_rev[]
```

6. RTCMP: Real Time Control and Management Protocol

6.1. RTCMP Functionality

RTCMP is responsible for run-time control messages or other information regarding a channel that are to be transmitted by means other than the data path provided by the real-time channel itself. The RTCMP specification in this document is preliminary and subject to revision in later versions.

6.2. Interface with RTIP

In case of an error, RTIP or RTTP calls RTCMP with a message like this

```
        notify_error(desc, pkt, ERROR_TYPE, ERROR_CODE)
        rtipState desc;  /* the channel descriptor. */
        char *pkt;       /* packet with error */
        int error_type; /*  is the category of error indicated. */
        int   error_code;   /*   is   the   further   subdivision   of
ERROR_CODES.*/
```

RTCMP is responsible for ensuring that the path of a channel is available. It does so by sending "ping" messages to the adjacent RTCMP modules at regular intervals and verifying if the nodes reply back. At the sending end, RTCMP checks at regular intervals whether the sending process is alive.

The following categories of errors are reported to RTCMP by RTTP/RTIP.

`PERF_GUARANTEE_VIOLATION:` Performance guarantees of a channel are violated. It can be verified only if RTIP or RTTP or both are collecting statistics.

`ROUTER_FEED_BACK_INFO:` The sites at the ends of a network may wish to complain to a client if it is found to be violating its traffic rates.

`CHANNEL_MISSED:` when the channel used by a packet can not deliver the data due to a failure somewhere in the network.

The following actions are taken on each notify error-code and error type:

```
Event           Action
CHANNEL_MISSED
                hit RCAP panic button for the channel
PERF_GUARANTEE_VIOLATION:
                send notification to the source node
RATE_CONTOL_INFORMATION:
                Ignored right now
```

The following messages may be received by RTCMP

Message	Action
ping request	send ping reply back
3 pings lost	hit RCAP panic button for all channels on link.
ping reply	mark node as alive
ping nack	mark node/channel as dead

The format of a ping message is shown in Figure 6.

The timestamps in the ping message can be used to synchronize the clocks on adjacent nodes, if necessary.

The `PING_CODE` is zero.
The following REQ/REPLY code are used:
00000001 : Node is alive request
00000010 : Node is alive reply
00000100 : Connection is okay
00001000 : Connection is non-existent

Other types of RTCMP messages will be required to handle different kinds of errors that may arise. This subject is currently under investigation.

RTCMP messages may travel either over IP or over pre-existing RTIP channels. Other kinds of RTCMP messages will be needed to offer a more comprehensive control protocol. In the first

```
0         7        15            31
++++++++++++++++++++++++++++++++++
!PING_CODE! REQ/REPLY! checksum  !
++++++++++++++++++++++++++++++++++
! From node IP address           !
++++++++++++++++++++++++++++++++++
!       To node IP address       !
++++++++++++++++++++++++++++++++++
!       Unique Conn ID           !
++++++++++++++++++++++++++++++++++
! UniqueCnnID  ! unused          !
++++++++++++++++++++++++++++++++++
! ORIGINATE TIME_STAMP           !
++++++++++++++++++++++++++++++++++
! RECEIVE TIME_STAMP             !
++++++++++++++++++++++++++++++++++
! TRANSMIT TIME_STAMP            !
++++++++++++++++++++++++++++++++++
                  Figure 6
```

implementation, these messages will travel over IP.

7. Implementation Notes

In this document, we have defined the logical design of the RTTP and RTIP protocols. In an implementation, a number of optimizations can be done to make the protocols run faster.

The main optimization has to deal with buffer management. In our design, separate buffers have been allocated to RTTP and RTIP modules for the same channel. Since both RTTP and RTIP at the endpoints of communication are on the same machine, the buffer space can be shared between them, avoiding unnecessary copying.

Under this scheme, a buffer will be allocated from the RTTP/RTIP buffer pool whenever a `rttp_send` is done, and the buffer will be released when the packet is actually transferred on to the link. At the receiving end, a buffer will be allocated when a packet is received by RTIP and released when the user does a `rttp_receive` to get the message.

Another area in optimization is that of rate-control. RTTP at the sender is supposed to space out packets so that they always obey the traffic specifications. RTIP at the sender determines whether these traffic constraints are actually being obeyed. Since the mechanism to regulate as well as the mechanism to verify can be implemented easily by means of calendar queues, one of the operations can be avoided. We recommend that the RTTP at the sender does only a logical regulation, by passing a future `eligibility_time` to the RTIP module. RTIP deals with the packet as if it had been received at the instant of `eligibility_time`.

Yet another optimization can be done by using the channel descriptors instead of the channel identifier when RTTP calls RTIP and the sender and vice-versa at the receiver. This avoids the need to map from the channel identifier to the descriptor twice.

High speed communication should also be aided by the use of scatter-gather mechanisms. Thus, the send primitives in RTTP or RTIP should not expect the packet as a contiguous data array, but rather as a structure with pointers to different header structures in the protocol hierarchy. These headers could be put together at the instant when the final copy to the physical link is made.

Other optimizations which will speed up the implementation of the protocols should be suggested by experimentation in the future.