Design of a Continuous Media Data Transport Service and Protocol*

Mark Moran† Bernd Wolfinger‡

TR-92-019

April 1992

Abstract

Applications with real-time data transport requirements fall into two categories: those which require transmission of data units at regular intervals, which we call continuous media (CM) clients, e.g. video conferencing, voice communication, high-quality digital sound; and those which generate data for transmission at relatively arbitrary times, which we call real-time message-oriented clients. Because CM clients are better able to characterize their future behavior than message-oriented clients, a data transport service dedicated for CM clients can use this *a priori* knowledge to more accurately predict their future resource demands. Therefore, a separate transport service can potentially provide a more cost-effective service along with additional functionality to support CM clients. The design of such a data transport service for CM clients and its underlying protocol (within the BLANCA gigabit testbed project) will be presented in this document. This service provides unreliable, in-sequence transfer (simplex, periodic) of so-called stream data units (STDUs) between a sending and a receiving client, with performance guarantees on loss, delay, and throughput.

*This research was supported by the National Science Foundation and the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives, by AT&T Bell Laboratories, Hitachi, Ltd., Hitachi America, Ltd., Pacific Bell, the University of California under a MICRO grant, and the International Computer Science Institute. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing official policies, either expressed or implied, of the U.S. Government or any of the sponsoring organizations.

†The Tenet Group, Computer Science Division, University of California, Berkeley, California 94720 and International Computer Science Institute, 1947 Center Street Berkeley, California 94704, moran@tenet.berkeley.EDU

‡University of Hamburg, Computer Science Department, Bodenstedtstr 16, D-2000 Hamburg 50, wolfinger@rz.informatik.uni-hamburg.dbp.de

Table of Contents

1. Introduction

Client applications, hereafter referred to simply as *clients*, with real-time data transport requirements fall into two categories: those which require transmission of data units at regular intervals (hereafter referred to as *continuous media* or CM clients), and those which generate data for transmission at relatively arbitrary times (hereafter referred to as *real-time message-oriented* or simply *message-oriented* clients). Examples of the former are video conferencing, in which video frames (of fixed or variable length) are sent from source to destination once per frame time (e.g. 33 ms); voice communication, in which one sample is transmitted for every sampling period (i.e. 125 μ sec); playback of high-quality digital sound, which also must transfer a fixed number of samples per second; and transmission of sensor data which is measured and transferred with strict periodicity. Examples of real-time message-oriented clients are those which require urgent messages, such as may be needed for building geographically distributed applications; process control applications; and a mail service with guaranteed delivery latency.

It is generally accepted that dedicated transport protocols are necessary for high speed networks. Adaptation of existing transport protocols, originally designed for lower speed networks (such as OSI Transport Protocol Class 4 or TCP), to high speed environments is not straightforward, and may not provide satisfactory performance to transport service users of future networks. Therefore, considerable research has been conducted in the design of completely new transport protocols to support high speed end-to-end communication between users. Surveys of the general requirements these protocols must satisfy, and of existing protocol proposals can be found in e.g., [DDK90], [LaS91], [WrT90], [Zit91]. In these publications, it is suggested that new algorithms are needed to support basic transport protocol functionality (such as flow control, error detection and correction, connection management, etc.). In addition, the use of specific implementation techniques, e.g. parallel processing, is advocated. Current transport protocols designed for high speed networks include, e.g., Delta-t Transport Protocol, cf. [Wat89], Network Block Transfer Protocol (NETBLT), cf. [CLZ87], Versatile Message Transaction Protocol (VMTP), cf. [ChW89], Express Transport Protocol (XTP/PE), cf. [Che88], and the protocol designed by Netravali et al. and described in [NRS90].

The literature suggests general agreement among network designers that transport protocols should be tailored to meet the various transport service requirements of end users. Requirements of various users can be supported by using rather general transport services and providing options to flexibly adapt the service to the differing requirements of users (e.g. during establishment of a transport connection). On the other hand, it is also possible to split the transport service *a priori* into two (or more) different, cleanly separated, services. Each service would support a class of users with similar data transport requirements. This second solution is chosen in this paper, as we will describe a transport service designed for continuous media clients, which we expect to coexist with a transport service designed for message-oriented clients. For surveys on the requirements of continuous media applications for data transport, the reader is referred to [HSS90] and to [ITC91]. Requirements of video transfer, in particular, can be found in [LiH91]. The remainder of this section details the arguments in favor of a separate transport service for CM clients.

Because CM clients are better able to characterize their future behavior than message-oriented clients, a data transport service tailored for CM data can more accurately predict the future resource demands of such clients and potentially provide them with a more cost-effective service.

In addition to the predictability of data transfer times mentioned above, CM clients have data transfer requirements that cannot be *efficiently* met by a service designed for message-oriented clients. One such requirement is the abstraction of *logical streams.*[1] Because of the periodicity of CM data transfer, data can be logically delineated by *time* as well as by buffer location (e.g. one video frame corresponds to the data offered for transmission during a specific 33 ms interval). A *(logical) stream* then consists of a time sequence of such data units. By making such streams visible to the data transport service, network and system resources can be conserved between streams (e.g. reserved buffers can be ‘‘loaned’’ to other traffic until a new stream is started). More importantly, some connection

[1] It should be noted that the ‘‘streams’’ defined in this document are not related to Unix (AT&T System V) streams.

parameters may be redefined for the duration of a stream, allowing resources to be conserved and providing better coordination between sender and receiver. For example, one technique for implementing ‘‘freeze frame’’ and ‘‘slow motion’’ in a video playback application would be to stop the current stream, and to start a new, slower, stream. Use of logical streams can also simplify the synchronization of data from separate connections at the receiver.

Another important difference between message-oriented and CM clients is their error-handling requirements. A message is generally thought to be of no use if any part of it is lost, so the entire message is often discarded in the case of a partial loss of data. In addition, most message-oriented clients cannot use corrupted data, so corrupted data is discarded. Most CM clients, however, can recover from partial data loss, and therefore desire all data which has been correctly received by the data transport service to be delivered to the application, even if some data is lost. In addition, many CM clients can utilize corrupted data. CM clients would, therefore, prefer a service that gives an indication of lost data, delivers all correctly received data, and optionally delivers corrupted data (along with an indication that the data has been corrupted). Of course, a message-based service could also implement these error-handling requirements, but they would probably not be useful to traditional message-oriented clients, which normally require uncorrupted data.

The last difference between CM and message-oriented clients discussed here is the nature of the entity generating the data. For applications that send messages, the data is always generated by a client process, which can initiate transmission of the data by contacting the data service. However, for many CM clients, the data is provided for network transfer by a ‘‘dumb’’ hardware device, such as a hardware coder-decoder (codec) for compressed video. Therefore, an additional user-level process would have to intervene to schedule and initiate data transmission.

The above differences in data transport requirements between message-oriented and CM clients justify a dedicated CM data transfer service to provide better service for CM applications in four ways: (1) a better traffic model for characterizing CM traffic and specifying performance requirements; (2) the abstraction of (logical) streams, which are visible to the transport service; (3) CM specific error handling including delivery of all good data received and (possibly) corrupted data, and of optionally replacing corrupted/lost data with dummy data; and (4) the elimination of the need for a synchronous rendezvous (e.g. via a system call) between a client process and the service for the transmission of each piece of data.

At this point, we would like to emphasize that there is no fundamental reason a message-oriented transport service could not offer a service that included (1) - (3); however, as we argued above, these capabilities are neither required nor desirable for most message-based applications, and hence it seems wiser to implement a new service to provide them to CM clients. These CM specific capabilities will now be discussed in more detail.

(1) Better characterization of CM traffic and performance requirements. Any service which provides real-time guarantees will require a client to characterize its traffic and specify its performance requirements. For a standard message-based service, these specifications will be made in terms of messages. One possible interface which provides a good model for message-oriented applications is described in [FeV90] and is repeated below.

- Traffic parameters:

  $s_{\max}$: maximum message size
  $x_{\min}$: minimum spacing between messages
  x_{ave} : maximum value of the average spacing between messages
  I : averaging interval for x_{ave}

- Performance parameters:

  D: maximum end-to-end delay
  J: maximum end-to-end delay jitter
  Z: lower bound on probability of satisfying the delay guarantee
  U: lower bound on probability of satisfying a probabilistic jitter guarantee
  W: lower bound on probability of delivering messages correctly

These parameters would then be translated into parameters based on network-level packets to implement the message-oriented service (cf. section 3.1).

This characterization provides a good model for an application which sends messages of relatively fixed-size at somewhat arbitrary intervals (limited by $x_{\min}$ and x_{ave}). The characterization of burstiness in data rate as a variation in the

speed of sending fixed-size messages seems to be common to all message-oriented traffic models with which we are familiar, e.g., [Cru87], [AHS90]. However, this characterization does not provide an adequate model for isochronous (CM) applications, which transmit a variable amount of data at fixed intervals. A canonical example of such an application is a client transmitting a compressed video stream. The application might naturally choose to transmit each (compressed) video ‘‘frame’’ as a message. In the characterization above, we would have $x_{\min} = x_{ave}$ and there would be no way of characterizing the burstiness of the data stream caused by the variable compression from one frame to another. Such a situation could greatly overestimate the network resources required if the compression ratio were highly variable, such as when both inter-frame and intra-frame compression are used. [e.g. Leg91] Therefore, most message-based traffic characterizations are inadequate and will cause more resources to be reserved for the client than are needed to meet its performance requirements.

This problem could be remedied by providing an interface that allowed clients to model their traffic as a fixed interval between messages ($x_{\min}$), and a maximum ($s_{\max}$) and average message size (s_{avg}). The translation from these parameters to those defined for network packets is straight-forward. However, such an interface would *not* provide a good model for characterizing the traffic of most message-oriented applications. Therefore, to adequately characterize both message-oriented and CM traffic, two models would be required. In this proposal, we will provide a characterization similar to the one already indicated here, but using parameters which better characterize a stream of continuous media data (and hence allow more efficient use of network and host resources).

(2) Logical stream abstraction. The data transmission of a CM client can be modelled as a series of *logical streams* as described above. The use of logical streams allows the sender to logically partition the data traffic on a network channel over time. (In this document, we will use *channel* to refer to a simplex connection for which network and system resources have been allocated in order to provide performance guarantees.) Because streams are seen by the transport service, both the sending and receiving host and (possibly) the network can allow resources allocated to the channel to be used by other traffic during streams with looser performance requirements, and between streams. It should be noted that in order to conserve resources in the transport level (or below) the streams must be visible to the transport level, and hence cannot be provided as a session level service. The stream abstraction also allows closer coordination between the sender and the receiver. For example, a source may wish to inform the receiving client of different data rates for different phases of a conversation (e.g. as described above for ‘‘freeze frame’’ and ‘‘slow motion’’ in a video playback application). Using streams would be preferable to releasing and re-establishing a channel because defining a new stream would incur less overhead and latency, and because of the possibility that the establishment of the new channel could fail (assuming channels cannot be modified). Since the underlying channel is not altered (i.e. no new resources are allocated), only some of the channel parameters can be respecified for the duration of a stream and these can only be made less strict (e.g. lower data rate).

Streams could also be used (e.g. with timestamps) to synchronize two or more channels, or to indicate to the receiving client a change in the final destination of the data (e.g. switch output files). Since streams can be of any arbitrary length and should be visible to the data transport service, they could not be adequately supported by a message-based transport service.

(3) CM specific error-handling. Because many continuous media clients interact with human users, they can often tolerate some loss of transmitted data without suffering a perceived loss in service quality. Traditional message interfaces are designed for transfer of file-like data which cannot be used if it is not completely free of errors. The strict delay requirements of real-time traffic do not allow for sophisticated retransmission strategies, so messages in which some data is lost or corrupted will simply be discarded.[2] A service geared toward continuous media should provide a mechanism for delivering all data received by the transport service to the user with an indication of any data lost because of corruption or failure to arrive on time.

In addition, many CM applications can actually perform better if corrupted data is returned (e.g. uncompressed video) or if some ‘‘dummy’’ data is substituted for lost or corrupted data (e.g. [HTH89]). Such error handling capabilities would have limited utility for message-oriented applications.

---

[2] Even for those applications in which delay requirements are loose enough that retransmissions could be at-

(4) No explicit interaction required for transfer of data. Perhaps the greatest limitation of a message-based service for CM clients is the need for a user-level process to contact the service in order to initiate each data transmission (i.e. to call *send*). There are two problems with this requirement:

- It introduces unnecessary system overhead due to the requirement of a synchronous *rendezvous* between the data transport service and the client for each data send. (unnecessary because the time of the next *send* is well-known for isochronous traffic!)
- It requires a client process to intervene between data generation and data transmission.

If data is being generated by the client process, the second requirement is not a limitation, but if the CM data is generated by a hardware device (e.g. video coder-decoder) or from a continuous media I/O server such as is described in [AGH90], the client process adds programming complexity as well as latency and system overhead.

The most natural interface to such ‘‘dumb’’ clients is a shared circular buffer with a small amount of (shared) state describing the amount of data in the buffer. This is the service we propose for CM. Library routines may be implemented on top of this service to perform all synchronization tasks and provide a more familiar *send* and *receive* interface for client processes that generate their own data. The library routines that implement the *send* and *receive* would operate as user-level processes on behalf of the client. Therefore, even for these clients, system overhead would be reduced since a user-level process would not need to cross a protection boundary into the kernel (e.g., via a system call) after channel establishment; all interactions would take place through the shared state.

Anderson et. al have developed a continuous media I/O server that utilizes the additional information associated with a CM stream to provide a simple and efficient interface to a file system for constant rate applications [AGH90]. The additional information implicit in an isochronous stream of data can also be used in providing better functionality and more efficient service for variable-rate applications transmitting data over a network which provides real-time guarantees. This document proposes a data transport service for variable-rate continuous media clients to be implemented on top of a real-time network service such as that described in [FeV90]. The service we propose provides better functionality for continuous media clients without sacrificing efficient utilization of network resources.

The service requirements to be supported by the CM Transport Service, presented in this paper, are motivated and introduced in detail in section 2. Section 3 describes the assumptions we have made regarding the functionality of the underlying network service and the environment in which the service will be provided. Section 4 defines abstract service primitives and uses them to present a more formalized service description. Section 5 describes the functionality of the continuous media transport layer and the protocol defined for providing this functionality. Section 6 gives implementation considerations.

2. Description of the Continuous Media Transport Service

The CM transport service described in this document provides unreliable, end-to-end, sequenced, periodic transfer of *stream data units* (STDUs) from a sender to a receiver (simplex) with guaranteed performance. An STDU is a data unit for which the client wishes the service to maintain boundaries. If a client specifies that all STDUs are one byte in length, the stream becomes a byte stream. The CM transport service packetizes STDUs and schedules packet transmission to efficiently utilize network resources while meeting performance requirements. It also provides the receiving client with an indication of lost or corrupted data. In this section we will first describe the transport requirements of several sample CM clients. Then we will describe our service in more detail and demonstrate its use for our sample clients.

2.1. Examples of CM clients

We now describe several representative CM clients to motivate the design for our proposed CM data transport service. All the applications listed have strict delay requirements (which are stated for each) in that data must be available by the time it is needed, and strict delay jitter requirements in that data cannot arrive so early that it overflows buffers allocated to the client.

---

tempted, we expect that the amount of data which would need to be stored at the sender to enable it to retransmit data across a high bandwidth-delay product network would be prohibitive.

Since excessive delay jitter (i.e. delivery too early) can be absorbed by buffering in the service provider, delay jitter is not listed as a requirement for the clients. However, as described in the section on implementation considerations, compensating for delay jitter can add to the end-to-end delay of the channel and requires additional buffer space to be allocated to the channel on the receiving end-system.

2.1.1. Video conferencing

To simplify our example, we will only consider transmission of the video portion of a video-conferencing stream. The stream is characterized as follows:

A. Video only, 1/4 screen (320x240, 24 bits per pixel = 1.84 Mbit or 225 KB per frame).

B. 15 frames per second.

C. Minimum compression without inter-frame coding is 20:1.

D. Minimum long-term average compression using inter-frame coding is 50:1. We are assuming no scene changes in video conferencing streams, so we can assume we get ‘‘average’’ compression over less than 8 frames.

E. The compression algorithm uses inter-frame coding most of the time. Every 9th frame (600 ms), however, is sent with no inter-frame dependencies so the receiver can recover from data losses.

F. One-way delays of less than 300 ms are acceptable.

G. Corrupted data cannot be used.

H. We assume that the compression algorithm can maintain acceptable image quality if at least 90% of the data for each frame is delivered.

I. Video frame boundaries are delineated within the data stream, and the compression algorithm has mechanisms for recovering after data loss; therefore, there are no boundaries in the data which need to be maintained by the data transport service.

J. We will also assume that the compression algorithm recognizes some code (we shall assume 16 or more consecutive 0’s as a convenient code) to indicate that a data loss of unspecified length has occurred at that point in the stream.

From this description, requirements can be extracted which the client would like to communicate to the data transport service:

- Periodicity of video frames will be 66.7 ms. (from B)
- No video frame will require more than 11.2 KB. (from A,C)
- The mean size of compressed video frames (measured over a 600 ms interval) is bounded above by 5.2 KB. (from A,C,D,E)
- Delay of the stream should not exceed 300 ms. (from F)
- Corrupted data should be discarded and considered lost. (from G)
- The granularity of data loss should not be greater than 0.52 KB. (from A,C,D,E,H)
- There is no need to maintain boundaries of video frames in packetization, i.e. byte-stream transfer can be used. (from I)
- A special code (consisting of at least two bytes of zeros) should be inserted in the data stream whenever data is missing for some reason. (from J)
- The video coder/decoder (codec) is a hardware device which is capable only of DMA transfer into a client’s address space. A separate control program is available to handle control functions from time to time.

2.1.2. Multiplexed voice

Another client would like to send PCM voice:

A. Each sample is 8 bits (uncompressed).

B. 8000 samples per second.

C. Delays of up to 300 ms can be tolerated (cf. voice channels via satellite links).

D. We will assume that loss of more than 100 consecutive samples (12.5 ms) per second produces intolerable degradation.

E. The application can use an indication of data loss to recover by interpolating for the lost data.

The interpretation of this characterization is straightforward.

2.1.3. Graphics/image window

Another type of stream that has much looser real-time requirements is the transmission of slowly changing images. ‘‘Slowly changing’’ could mean that only a small part of the image is changing (e.g. someone writing on a ‘‘notepad’’ or ‘‘blackboard’’) or that the complete image is replaced from time to time (e.g. slides). Such a stream could have the following characteristics:

A. Images are full screen NTSC TV quality (640x480x24 = 900 KB)

B. Delay is not important (unless window is synchronized with another stream)

For ‘‘notepad’’ images:

C1. Difference-frame encoding can be used to get very high compression (probably well above 100:1)

D1. Frequent updates are desirable (at least 2 per second)

For still images:

C2. Low-loss compression can be obtained at a minimum compression ratio of 10:1

D2. Much slower rate of update

In general, transmission of still images does not map well onto a continuous media service, except in the case (described here) in which images are sent with some regularity or the case in which the images have to be synchronized with a continuous media stream such as audio. If we wanted to handle both of these types of images on the same channel, we would have to use variable-size data units and inter-frame coding. In this case, we would like to communicate the following requirements to the transport service:

- Periodicity is 0.5 seconds (C1)
- Maximum size of a compressed frame is 90 KB (A,C2)
- ‘‘Average size’’ of a compressed frame is 13 KB computed over 10 seconds (A,C1,D1,C2,D2: 13 KB is the average size of an update for the notepad service, assuming a completely new ‘‘page’’ every 10 seconds. For still images, we divide the maximum image size (90 KB) by this average to get the maximum transmission time for each image (in this case about 7 periods or 3.5 seconds)
- Delay is not important (B)

2.1.4. Periodic sensor data

A client that periodically sends sensor data to a central location (e.g. control of a nuclear power plant) could advantageously use a CM data transport service. The size of messages would probably be constant, and high reliability would be desired. Incorrect data could not be used, and some indication would have to be given that data was missing. The characterization would not add anything not already covered above and hence is omitted in the interest of brevity.

2.1.5. Multimedia International Classroom

The final example demonstrates the needs of clients that require synchronization among a number of streams and those that may wish to multiplex a number of streams over time on the same channel. The application transmits a recorded or live lecture to remote locations and optionally records the lecture at any of the sites. Each receiving location has at least two display ‘‘windows’’: one on which moving images are displayed, and another for slowly changing images which may be graphics (e.g. transparencies or live handwriting on a ‘‘notepad’’ as described above) or still images of TV picture quality, which will be changed infrequently, i.e. less than once per 4 seconds. The application can support interactive lectures in which students are allowed to interact vocally with the instructor. In this case, a return voice channel is established from each site to the source of the lecture where it is merged with the audio from the source. Each site may optionally receive the instructor’s lecture in another language, and sub-titles for recorded movies.

Since only a subset of the streams described above need to be transmitted at any one time, network resource allocations can be minimized if these streams are multiplexed over time onto a smaller number of network channels. Therefore, only certain combinations of the above streams are supported, and a class meeting consists of a number of consecutive *phases*. Switching between phases is controlled from the source via a simple menu and offers the following possibilities:

- *Lecture mode:* Sending video of the instructor in the video window along with 128 kbps for the instructor's audio and an additional 128 kbps for a translation. The second window is handled as the slowly changing still image window described above, supporting a notepad or TV still images. Both the instructor's video and the image window are synchronized with the instructor's audio.
- *Movie mode:* Sending a moving video with a stream of low-quality stereo sound (128 kbps per audio channel) and the option of textual subtitles in up to two other languages. The video stream is synchronized with any audio streams and the subtitle stream if it exists.

These phases can be supported by five channels of four separate types. The requirements of each of these types is listed below:

1. *Video channel:* This channel will be used for transmitting either video of the instructor in *lecture* mode or the moving video in *movie* mode. This channel is the same as the video-conferencing example above except that in *movie* mode the image is four times bigger, will change at 30 frames per second, and may experience up to one scene change per two seconds. This channel must be synchronized with the audio and/or images on the other channel.
2. *Audio channels (2):* Two channels will be established, each of which carries 128 kbps audio. In *lecture* mode one audio channel is used for the instructor's audio and one for a possible translation. In *movie* mode, each channel will carry audio for the movie. These channels must be synchronized with the video window. Each channel has the following requirements:
   - Periodicity is 12.5 ms (100 samples of 16 bits each per period, chosen as a reasonable tradeoff. A small period is desirable to limit the amount of data lost in a burst, but a large period is desirable for efficiency.)
   - Constant amount of data sent per period is 200 bytes
   - Maximum granularity of data loss is 200 bytes
   - Indication of data loss is required
3. *Notepad/Image Channel:* This channel is exactly as described in the previous example. In *lecture* mode, it will carry either a notepad (i.e. ''blackboard'') of color graphics, or a still image that takes about 10 seconds to change. In *movie* mode, it will carry two streams of subtitles for the movie. This channel must be synchronized with the video channel.
4. *Return channel:* This channel is only required for interactive classrooms. It transmits 128 kbps of audio from each site back to the source.

2.2. Definition of the CM data transport service

We now define a service designed to meet the needs of the types of CM clients described above. As stated previously, the Continuous Media Transport Service (CMTS) provides unreliable, sequenced transport (simplex, periodic) of stream data units (STDUs) between a sending client (C_S) and a receiving client (C_R), with performance guarantees on loss, delay, and throughput. The service optionally replaces each lost or corrupted piece of data with a *dummy* piece of data specified by the client during channel establishment. Data is passed from C_S to a CMTS entity operating on the same end-system (CM_S) via a shared circular buffer. Synchronization between C_S and CM_S is provided via traffic and performance parameters established by both parties at the beginning of the conversation, and through explicit synchronization variables. The relationship between these entities is shown in Figure 2.1. (Refer to the section on implementation for more details.)

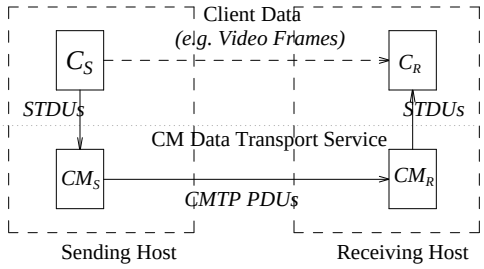


Figure 2.1: Relationship between CM and client entities

All traffic and performance parameters are defined in relation to two basic units: the stream data unit (STDU) and the periodicity of the

conversation (*T*). As previously stated, an STDU is a data unit whose boundaries must be preserved by the CMTS and indicated to C_R. C_S decides how the stream to be transferred to C_R is mapped onto a sequence of STDUs as illustrated in Figure 2.2. Typically, for a CM application, the information to be transmitted (e.g. voice signal, sequence of images) is first digitized by a coding process (or possibly a sequence of such processes). The coding process maps a continuous *signal function* (in the sense of coding theory, e.g. voice, moving scene) onto a sequence of *code words* (cf. coding theory again). A CM application then has several options in mapping these code words into a sequence of STDUs:

a) Sequence of code words mapped onto sequence of bytes (byte stream), 1 byte corresponding to 1 STDU;

b) one-to-one mapping of code words onto STDUs;

c) concatenation of several code words to build one STDU, e.g. in the case, where different sub-streams are multiplexed (time-multiplex) by an application to form one overall stream;

d) combination of a code word and a time-stamp into one STDU (where a ''time-stamp'' is some reference to the time interval the code word represents); as would be required in transmitting a stream, which had been stored, along with information to reconstruct its original timing.

The periodicity, *T*, of a stream can also be specified by an application. The periodicity characterizes the frequency of coding events (typical values chosen for periods would be $T=k\times0.125$ ms in PCM-voice coding or $T=k\times33$ ms in transfer of a video stream, where k is an integer). The data corresponding to a time interval of length *T* then maps into an integral number of STDUs. The CMTS service recreates on the receiver, the stream that had been seen on the sender at the granularity (in time) of a period, *T*. If the time on the sender is denoted by *t*, and the time on the receiver is likewise denoted as τ, then this definition of the service implies that data corresponding to an interval Δt_i at the sender must arrive before the beginning of the corresponding interval $\Delta\tau_i$ at the receiver.[3]

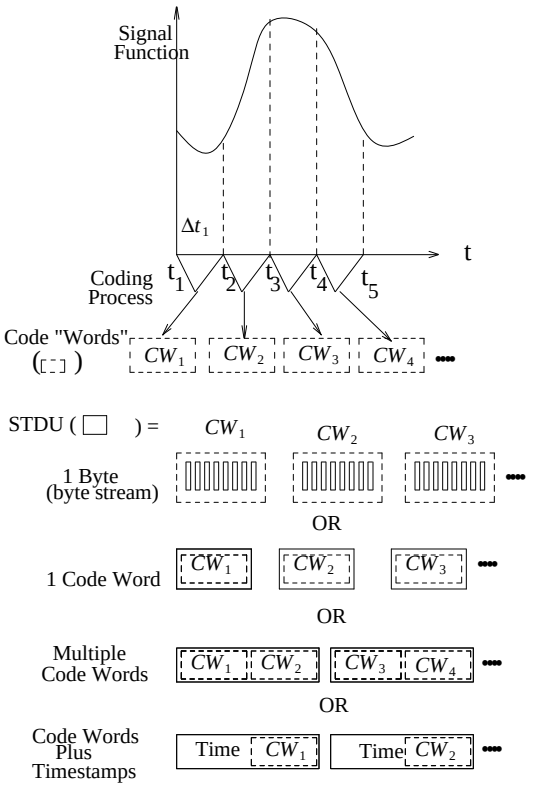


Figure 2.2: Mapping of CM data into STDUs

Figure 2.3 illustrates the basic timing within transfer of a sample stream. The time-shift between the stream as seen at the sender and the same stream as seen at the receiver is defined to be D_{stream} , the end-to-end delay of the stream.[4] If the starting time of the stream at the sender is denoted by $t=t_0$, and the starting time at the receiver is denoted by τ_0, then D_{stream} is given by the absolute difference in these starting times, or

$$D_{stream} = \tau_0 - t_{0,}$$

providing clocks are synchronized (i.e. read the ''same'' time). In order to maintain the continuity of the stream, all STDUs legally presented for transmission (i.e. without violating traffic characteristics) before the end of the interval Δt_i, where Δt_i is the interval $[t_{i-1}, t_i)$, must be available for C_R to read by time τ_{i-1}, where τ_{i-1} is the beginning of the corresponding period on the receiver given by

$$\begin{aligned}\tau_{i-1} &= \tau_0 + (i-1)T \\ &= t_0 + D_{stream} + (i-1)T.\end{aligned}$$

[3] Of course $|\Delta t_i| = |\Delta\tau_i| = T$, for all *i*, if $|I|$ denotes the length of interval *I*.

[4] If the sender and the receiver were on the same machine, the periodicity of the stream could be used to

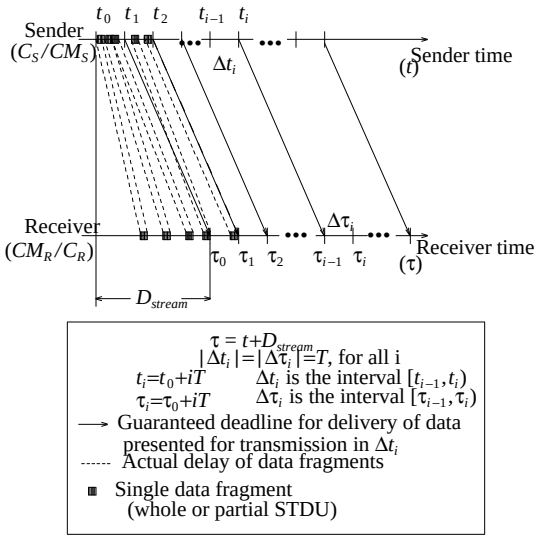


Figure 2.3: Timing diagram for Continuous Media Transport Service

C_S must inform CM_S of the beginning and end of each logical stream. As stated above, the model we are using for interactions between C_S and C_R is that the data generated during an interval Δt_i by C_S is needed by C_R during the corresponding interval $\Delta\tau_i$. Therefore, after indicating the beginning of a logical stream, C_S is *obligated* to ensure that all the STDUs which C_R will need during the interval $\Delta\tau_i$, are in the shared buffer before the end of the interval Δt_i on the sender. Because a sender may have difficulty presenting data within the correct interval (e.g. due to contention for the CPU or memory bus), another parameter, S_{Sslack}, is defined to be the number of bytes C_S is allowed to provide to CM_S *ahead* of schedule, i.e. before t_{i-1} for data which should be presented during Δt_i. (This concept of allowing some workahead for a sending CM client has been proposed by others, e.g. [And90].) C_S is also obligated to obey the traffic characteristics specified for the conversation and for the current stream. The obligations of C_S are summarized in Table 2.1.

At the beginning of a logical stream, C_S may define new traffic and performance parameters which will apply for the duration of the new stream only. These must be no more strict than the parameters of the actual channel. Beyond this restriction, the parameters of one stream have no relation to the parameters defined for other streams that will use the same channel (one at a time). The stream parameterization may be used to assist cooperation between C_S and C_R, since the characterization of a stream can more accurately represent current traffic and performance needs than the long-lasting channel parameters. The stream characterization may also allow system and network resources to be conserved (e.g. buffers could be ''borrowed'' from a channel which is between streams, although they will be reclaimed when a new stream begins). In addition, C_S specifies the startup delay ($d_{startup}$) and an upper bound on the duration of the stream (*duration*). $d_{startup}$ is the time between the indication from C_S that a new stream is about to begin (t_{start}) and the actual start of the stream (t_0). The interval between t_{start} and t_0 is used by C_S to preload the shared buffer, subject to the maximum workahead indicated by S_{Sslack}. The startup delay is not shown in Figure 2.3, but would occur immediately before t_0 on the sender's time axis. Since $d_{startup}$ occurs before the stream actually ''begins'', it is not included in the overall stream delay, D_{stream} (cf. Figure 6.3). The upper bound (*duration*) on the duration is passed to the receiving client which may use it in allocating storage resources at the receiver. It is an optional parameter (the default value is 0, which means the lifetime of the stream is unbounded).

After being informed of the beginning of a new logical stream, CM_S is obligated to cooperate with CM_R to transfer all STDUs for any interval Δt_i to C_R soon enough to meet the performance guarantees specified for the current stream, provided that C_S keeps its contractual obligations. The obligations of CM_S are also listed in Table 2.1. Shared synchronization variables are required to recover from the possible failure of one of these entities to satisfy its requirements and because of variability in the amount of data transferred during each period.

The CMTS entity at the receiving end-system (CM_R) and C_R also interact via a shared circular buffer, stream parameters, and shared synchronization variables. CM_R must inform C_R of the beginning of a new logical stream, indi-

---

achieve a simple producer/consumer synchronization by delaying the stream seen by the receiver one period relative to the sender, giving $D_{stream} = T$.

Table 2.1: Obligations between C_S and CM_S

Entity	Obligations
C_S	• Ensure that all STDUs associated with any interval Δt_i are in the buffer prior to time $t=t_i$, where t_i is the time at the end of the *ith* interval, Δt_i. • Ensure that no more that S_{Sslack} bytes corresponding to intervals Δt_j, where j > i are in the buffer before time $t=t_i$. • Obey maximum and average throughput limits defined below.
CM_S	• Send data fast enough to ensure that at time $t=t_{i-1}$ there is enough room in the shared buffer for all the data which could legally be presented in interval Δt_i. This amount of data is determined either by the maximum amount of data which C_S may present in *any* period, or the amount of data which C_S may present in the next period without violating average throughput constraints. In either case this amount is known to CM_S. • Transfer data to CM_R fast enough to meet stream delay requirements.

cating *duration* and a modified $d_{startup}$. After that, CM_R is obligated to put data in the shared buffer before the beginning of the interval in which it will be needed (i.e. before time $\tau=\tau_{i-1}$ for interval $\Delta\tau_i$), and C_R is obligated to remove from the shared buffer all the STDUs corresponding to an interval before the end of that interval (i.e. before time $\tau=\tau_i$ for interval $\Delta\tau_i$). Since a receiver may have difficulty removing data exactly during its corresponding interval, the value S_{Rslack} is defined at the receiver to indicate how far C_R can fall behind without data being lost due to buffer overflow at the receiver. More precisely, C_R can leave up to S_{Rslack} bytes corresponding to times $\tau<\tau_{i-1}$ in the buffer if the receiver is in the *ith* interval, i.e., $\tau_{i-1}\leq\tau<\tau_i$. As at the sender, the first period is defined to begin at time $\tau_0=\tau_{start}+d_{startup}$, where $d_{startup}$ is a parameter of the stream as described above (and may have been altered by CM_R). In addition to the above requirements, CM_R must indicate data loss and corrupted data in a manner agreed upon at the beginning of the conversation. In the implementation sketched in section 6, a structured buffer is used to maintain STDU boundaries and descriptors are used to indicate errors. The contractual obligations between CM_R and C_R are listed in Table 2.2.

Table 2.2: Obligations between C_R and CM_R

Entity	Obligations
C_R	• Ensure that no more than S_{Rslack} bytes corresponding to any time $\tau<\tau_i$ remain in the buffer after time $\tau=\tau_i$, where τ_i marks the end of the *ith* period, $\Delta\tau_i$.
CM_R	• Put all data needed during interval $\Delta\tau_i$ into the buffer before the beginning of that interval, i.e. before time $\tau=\tau_{i-1}$. • Honor traffic characteristics of the current stream across the CM_R/C_R interface. • Inform C_R of data errors.

In order to meet guarantees regarding buffer overflow and starvation avoidance at the receiving client, all four entities must be involved in a handshake at the beginning of the conversation so that each may approve traffic and performance parameters. At this time some of the entities will reserve system and network resources to ensure that they will be able to fulfill their contractual obligations. This handshake is accomplished as follows: C_S presents a proposed set of parameters to CM_S. If CM_S accepts these, it passes them on to CM_R, with some possible modifications and additions. If CM_R accepts the (revised) parameters, it passes a (possibly) revised version of the original set to C_R. If C_R accepts the conversation request, it informs CM_R, who informs CM_S, who informs C_S. The parameters of the handshake before a conversation are described below for the C_S/CM_S interface. Parameters for the handshake between other entities are analogous to those described here.

Traffic parameters

The traffic parameters listed below were chosen to capture those traffic characteristics of continuous media traffic that have the greatest impact upon resource requirements.

$STDU_{max}$: Maximum size of an STDU (bytes), i.e. maximum size of a logical unit for which boundaries must be maintained. ($STDU_{max}$ = 1 is allowed as a special case, leading to a transparent byte stream data transfer).

CONST_SIZE: A boolean to indicate whether STDUs are constant length.

CONST_NUM: A boolean to indicate whether a constant number of STDUs will be sent each period. This parameter is only considered if *CONST_SIZE = TRUE*.

T: Periodicity of the CM data stream (μs).

N_{max}: The maximum number of STDUs associated with a period. (This parameter is ignored if CONST_SIZE = TRUE, since it is redundant with the information provided by S_{max} and $STDU_{max}$).

S_{max}: The maximum number of bytes required to represent the input signal in one period.

S_{avg}: Upper limit on the mean number of bytes per period required to represent the input signal.

I_{avg}: The size of any averaging interval (in periods) over which S_{avg} is calculated.

S_{min}: This parameter limits the burstiness of the stream by providing a lower bound to the *expected* transmission per period. Although less data may actually be generated by C_S during the period, the average rate control mechanisms will always assume that at least S_{min} bytes are sent during every period. (See discussion below for a more thorough description.)

S_{Sslack}: At the sender, S_{Sslack} specifies the maximum number of bytes which C_S is allowed to put in the shared buffer early, i.e. before the beginning of the period to which the data correspond. (At the receiver, S_{Rslack} specifies the maximum number of bytes which C_R may leave in the buffer after the end of the corresponding period without risking data loss. S_{Rslack} is a parameter of the C_R/CM_R interface, not of the C_S/CM_S interface.)

Buffer: A structure indicating the location of the shared buffer provided by the client (*Buffer.ptr*) and its size (*Buffer.size*).

The motivation for including T has already been described. $STDU_{max}$ is also an obvious choice since it determines the (maximum) size of a data unit for which logical boundaries must be maintained. *CONST_SIZE* and *CONST_NUM* allow the client two options in specifying a variable data rate: either a constant number of variable-size STDUs or a variable number of constant-size STDUs may be sent in each period. We contend that, if a client wishes to send a variable number of variable-size STDUs per period, then it is probably not a CM client. N_{max} specifies the maximum number of STDUs to be sent each period. For streams of variable-size STDUs (*CONST_SIZE = FALSE*), N_{max} is given as an input parameter; however, for streams of constant-size STDUs, the input parameter is ignored and N_{max} is calculated from S_{max} and $STDU_{max}$.

S_{max} is used in allocating buffer space for smoothing, fragmentation, and reassembly at the source and destination end-systems. S_{avg} and I_{avg} are used to determine the long-term average throughput requirements of the stream, which influence the resources required for network transmission. The burstiness of the stream also influences resource requirements in the network, so CMTS may choose to smooth transmission of bursty data over several periods to reduce burstiness.

S_{min} is included to provide a reasonable limit on the burstiness of CM traffic. The rate control mechanism will assume that *at least* S_{min} bytes were transmitted in every period. Therefore, if less data is actually transmitted in some period, Δt_i, the average number of bytes transmitted in *every* averaging interval that contains Δt_i will be less than S_{avg}. The potential benefit of including this parameter is demonstrated in Figure 2.4. The stream depicted in (a) is a stream in which the variability of the data transmitted per period exhibits its own (longer) periodicity. One common example of such a stream is a compressed video stream that consists of an initial frame compressed with intra-frame encoding, followed by several frames that achieve higher compression using inter-frame encoding ([e.g., Leg91]). Without the extra limit on burstiness provided by S_{min}, such a stream could not be differentiated from a stream that is much more bursty. For example, the original stream and the more bursty stream shown in (b) would have identical traffic characterizations. However, a suitable choice for S_{min} allow a model stream that more closely

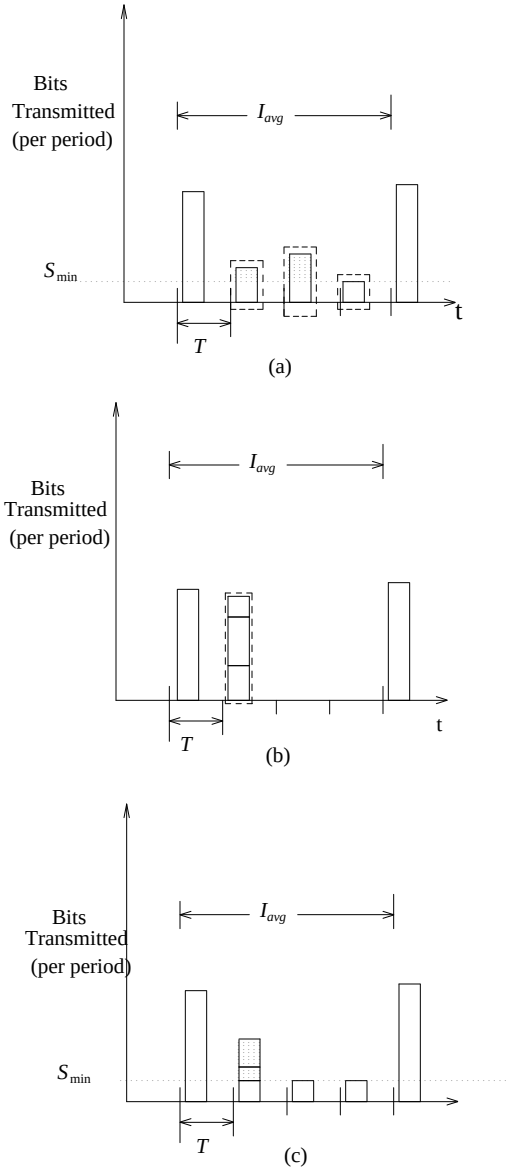


Figure 2.4: (a) Actual stream to be transmitted (b) Worst-case stream of model without S_{min} (c) worst-case stream of model including S_{min}

approximates the actual stream, as can be seen in (c). We would expect most applications to be able to define some non-zero minimum amount of data expected each period, since otherwise they would not be isochronous. This bound on burstiness provided by S_{min} is missing from other traffic models with which we are familiar. The inclusion of S_{min} and the characterization of burstiness via a variable message size (as opposed to fixed message size and variable rate of messages) more closely models the actual data production model of an isochronous client. Therefore, we expect such a model to better predict the resource requirements of CM streams and hence to allow more efficient allocation of network and end-system resources.

On most systems, memory allocation is best performed by the client, which then allows the service to share the allocated buffer. Therefore, the client must provide a structure (*Buffer*) giving the location and size of this buffer. User-level library routines can be provided to allow the client to query as to the minimum buffer size required for its requested channel.

Quality of service parameters

The parameters for indicating the quality of service (QOS) desired were chosen to sufficiently communicate the needs of most CM clients. The basic model of the CMTS service is that the stream on the sending end-system will be recreated on the receiving end-system at the granularity of a period. The service handles delay-jitter for the client (where delay-jitter is defined as the variability in delay) by ensuring the shared buffer at the receiving end-system is large enough to tolerate maximum possible jitter without overflow.[t] Because there are no explicit interactions between the client and the service, the implications of delay-jitter on timing of the stream do not apply.

Quality of service parameters for this model are described below.

D_{stream}: The maximum acceptable delay of the stream, where the delay of a stream, D_{stream}, is defined as the time between the the start of a stream at the C_S/CM_S interface (t_0) and the start of the stream at the C_R/CM_R interface (τ_0). D_{stream} implies a deadline for data arrival at the receiver, since all data associated with the ith interval on the sender, Δt_i, must arrive at the receiver before the beginning of the same period at the receiver, i.e. before τ_{i-1}.

S_{err}: The maximum *granularity* of data error due to either data loss or corruption (in bytes).

W_{err}: A lower bound on the probability that a unit of data transfer (of size $\leq S_{err}$) is correctly delivered to the receiving interface. It should be noted that this probability accounts only for losses due to dropped or delayed data. CMTS requires all packets on a stream to have the same value of W_{err}.

REPLACE: A boolean that indicates whether corrupted data should be replaced with dummy

data instead of being delivered as it is received.

Dummy: *Dummy* is a pointer to a buffer of size S_{err} which is used to replace each lost or corrupted *packet* if *REPLACE=TRUE*.

D_{stream} is the (constant) delay of each period, i.e. the actual time difference between the beginning of a period on the sending end-system and the beginning of the same period on the receiving end-system. This parameter is important to clients with latency requirements, e.g. interactive video conferencing. S_{err} and W_{err} describe the ''burstiness of errors'', since locality of errors is as important as frequency of errors for many CM clients. *REPLACE* and *Dummy* allow the client to specify a dummy packet to be substituted for lost and (optionally) corrupted packets. This service is useful for in-band signalling of data loss and for filling in holes in the data stream.

2.3. Use of service for sample CM streams

Since the multimedia international classroom application includes the other sample streams, it suffices as an example. We will calculate the relevant parameters for each of its four types of channels.

1. *Video channel:* We will use a byte-stream for the video channel, which can be written directly into the shared buffer by the codec (video coder-decoder) at the sender and read from the shared buffer by the codec at the receiver.

$$STDU_{\max} = 1$$
$$CONST\_SIZE = TRUE$$
$$CONST\_NUM = FALSE$$
$$T = 33.3\ ms$$
$$N_{\max}\ \textit{input parameter is ignored}$$
$$S_{\max} = 46{,}080\ bytes$$
$$S_{avg} = 18{,}400\ bytes$$
$$I_{avg} = 60\ periods$$
$$S_{\min} = 2000\ bytes$$
$$S_{Sslack} = \frac{S_{\max}}{STDU_{\max}} = 46{,}080 bytes$$
$$D_{stream} = 300\ ms$$
$$S_{err} = 1840\ bytes$$
$$W_{err} = 95\%\ (< 1\ packet\ per\ period)$$
$$REPLACE = TRUE$$
$$Dummy = 1840\ bytes\ of\ 0's$$

S_{avg} is larger for this video stream than for the video conferencing stream because this stream uses a larger image and anticipates more scene changes. I_{avg} is chosen as 60 periods because scene changes are expected, on the average, once every 2 seconds. $S_{\min}$ and S_{avg} should be measured for typical streams; here we will assume a reasonable value for both. $S_{\min}$ is not an absolute minimum, but an indication of the minimum amount of data to be expected per period if the average data rate is fully utilized. Although some periods may generate less data than $S_{\min}$, any averaging interval containing them would also then give an average less than S_{avg} because of the manner in which $S_{\min}$ is defined (and enforced). *Dummy* is chosen to indicate lost data to the codec.

2. *Audio channels:* The audio channels will be like voice channels, except as noted above

$$STDU_{\max} = 200\ bytes$$
$$CONST\_SIZE = TRUE$$
$$CONST\_NUM = TRUE$$
$$T = 12.5\ ms$$
$$N_{\max}\ \textit{input parameter is ignored}$$
$$S_{\max} = 200\ bytes$$
$$S_{avg} = 200\ bytes$$
$$I_{avg} = 1$$
$$S_{\min} = 200\ bytes$$
$$S_{Sslack} = \frac{S_{\max}}{STDU_{\max}} = 1$$
$$D_{stream} = 300\ ms$$
$$S_{err} = 200\ bytes$$
$$W_{err} = 99\%\ (lose \le 1\ packet\ per\ period)$$
$$REPLACE = FALSE$$
$$Dummy = null\ (do\ not\ replace\ lost\ data)$$

3. *Notepad/Image Channel:*

$$STDU_{\max} = 1\ byte$$
$$CONST\_SIZE = TRUE$$
$$CONST\_NUM = FALSE$$
$$T = 0.5\ sec$$
$$N_{\max}\ \textit{input parameter is ignored}$$

$$S_{\max} = 90\ KB$$
$$S_{avg} = 13\ KB$$
$$I_{avg} = 20\ periods$$
$$S_{\min} = 0$$
$$S_{Sslack} = 90\ KB$$
$$D_{stream} = 300\ ms\ (video\ stream)$$
$$S_{err} = 1300\ bytes$$
$$W_{err} = 95\%$$
$$REPLACE = TRUE$$
$$Dummy = all\ 0's$$

4. *Return channel* is the same as the audio channels

3. Requirements of Underlying Service and Execution Environment

Before the function of the CM layer can be described, assumptions about the underlying service and the execution environment must be stated.

3.1. Underlying Data Transfer Service

The CM layer will be built upon a internetwork layer data transfer service. CM requires the following minimum functionality from the underlying data transmission service:

- At least simplex data transmission
- Guaranteed minimum throughput
- Guaranteed maximum end-to-end delay
- Guaranteed maximum packet loss rate
- Lower layer returns some mechanism for discovering lost and mis-ordered packets (e.g. sequence number)

In addition to these requirements, the following characteristics are highly desirable (either for an easier/more efficient implementation or for added functionality):

- Corrupted data returned by underlying service (The functionality of CMTS is reduced without this capability, but it is not a fundamental requirement.)
- Underlying service indicates when it is returning corrupted data
- Underlying service indicates when data is lost
- Delay jitter can be limited by the underlying layer
- Underlying service can assemble packets for transmission from separate buffers, i.e. headers and data do not need to be contiguous in the same buffer
- Underlying service performs smoothing on packets delivered in bursts, e.g. leaky bucket. (We should note that the underlying service provides smoothing of packet transmission times, whereas CMTS provides smoothing of the data rate, i.e. a large burst is smoothed over a number of periods. CMTS must provide this smoothing of the data rate so that data can remain in the shared buffer and slow down a client that is sending too fast. However, smoothing of packet transmission times is best performed by only one layer, hence the lower level service.)
- Underlying service tries to schedule packet transmission according to times passed in by CM

The underlying service for network transmission in our prototype will be the *Real-Time Internet Protocol* (RTIP), which is described fully in [ZhV91]. RTIP provides all the required functionality and also returns corrupted data without indicating whether data is corrupted (it does not protect the data field), can limit delay jitter, and performs smoothing of packet transmission according to times passed in by the upper layer. RTIP clients describe their traffic characteristics and performance requirements using the following parameters.

Traffic Characteristics:

$s_{\max}$:
Maximum size of data to be sent on the internetwork channel (i.e. maximum size of a internetwork service data unit)

$x_{\min}$:
Minimum spacing between packets

x_{ave}:Upper limit on the average spacing between packets

I: Averaging interval for x_{ave}

Performance Requirements:

D: End-to-end delay bound

Z: Lower bound on the probability that a packet arrives within its delay bound

J: Delay-jitter bound (i.e. bound on the spread between the minimum and maximum delays of all packets sent on the channel which arrive

within the delay bound)

U: Lower bound on the probability that a packet satisfies its jitter bound

W: Lower bound on the probability that a packet arrives at the destination (i.e. is not dropped due to buffer overflow)

The send interface provided by RTIP is as follows (from [ZhV91]):

```
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. `eligib_time` is the time at which RTIP will assume the data was presented for delivery, although it may have actually been presented earlier. This allows the upper layer to use the traffic smoothing of RTIP instead of implementing its own timers for smoothing. `higher_level_encaps` can contain some information from the upper layer which needs to be carried on a per-packet basis. Although this feature is mentioned in the original RTIP design, it is not expected to be supported in future designs (including the prototype implementation), and hence will not be used by CMTS.

RTIP calls the function `rtip_indication()` to inform the upper layer at the receiver that data has arrived. This function is provided by the upper layer (i.e., by CMTS). Its syntax is as follows:

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

where `lcid, pkt, len, and higher_level_encaps` are the same as described above for `rtip_send()`; `pkt_seq_no` is the sequence number of the current packet. It is provided to allow the upper layer to detect missing packets (mis-ordered packets are discarded by RTIP).

RTIP channels will be established and administered by the *Real-Time Channel Administration Protocol* (RCAP), described in [BaM91a]. RCAP establishes channels and manages resources to ensure that RTIP can meet the guarantees made to its clients.

3.2. Execution Environment

The CMTS requires certain functionality from the operating system and hardware upon which it is running. Implementation of full functionality is not possible if the following requirements are not met:

- Pages of virtual memory can be locked into physical memory (for the shared buffer).
- CM can share memory pages with clients without requiring that the clients' execution state be loaded in order for shared pages to be accessed.
- A real-time clock with high precision timer (typically about 1 msec) is available. (We expect the requirements of the underlying service to be more strict in this regard.)
- Real-time scheduling of CPU and network is available, i.e. the latency between the time the CM process requests to run and the actual time the process runs can be bounded by a ''reasonable'' value. The same holds for network transmission.
- End-systems can provide a guaranteed minimum amount of processing time to CM during a specified interval.
- Similar guarantees on latency and CPU time are also provided for CM clients.
- Clock drift between communicating hosts is not ''too great''. (If delay requirements and buffer availability allow, clock drift can be tolerated by the workahead given by S_{Sslack} and S_{Rslack}.)

As with the underlying service, there are also several characteristics which would be advantageous, but are not strictly required:

- Clocks on communicating hosts are synchronized. (Additional buffer space and part of the end-to-end delay can be used to tolerate lack of synchronization between clocks.)
- Virtual copying of memory pages can be implemented efficiently via page re-mapping.
- Memory sharing can be controlled on a per-page basis.

We have not yet found a Unix-like operating system that provides all the features we require without prior modification. Work is currently in progress to modify Ultrix (DEC) and HP-UX (HP) to support our real-time protocols.

4. CM Service Primitives and Formalized Service Description

This section introduces abstract service primitives and specifies their intended usage to more formally define the CM data transfer service. First, we will define the primitives for channel establishment and their use. Then, we will define primitives for sending and receiving streams of CM data and their use.

4.1. Service primitives for channel establishment and tear-down[5]

To transmit CM data from C_S to C_R with performance guarantees, a connection must be established between these entities, and network and system resources must be allocated to this connection. As previously stated, such a connection with associated resource allocations is termed a *channel.* Channel establishment for CM channels is managed by the *Real-Time Channel Administration Protocol* (cf. [BaM91a] and [BaM91b] for a more detailed description of RCAP).

In this document we will describe the calculations and reservations necessary for the CM layer, i.e. for transporting data from the sending client, C_S, to to the receiving client, C_R via a guaranteed real-time internetwork service. The internetwork service we will use is provided by the *Real-Time Internet Protocol (RTIP)* and is described in [ZhV91] and [BaM91a]. Although this document will describe the channel administration functions (including channel establishment and teardown) of the CM service, it should be remembered that this functionality will be provided by the RCAP module, which will also provide channel administration for the message-oriented service, RMTP (also described in [ZhV91]).[6] RCAP will need to translate traffic and quality of service (QOS) parameters from those specified at the CM/client interface to those supported by the internetwork service, and request both processing time and buffer space on both the sending and receiving end-systems for CM functions (i.e. fragmentation, and scheduling calculations). The primitives which support this service are described below.

[5] It is assumed that creation of continuous media channels can only be initiated by the source.

[6] It should be noted that the CM transport service will be implemented using the same real-time internetwork service primitives as are used by RMTP and described in [ZhV91]; hence, CM channels will not differ from message-oriented channels at or below the upper interface of the internetwork layer.

Data types are as described in [BaM91b]. For convenience, the description is repeated here. Unless otherwise noted, the RCAP user interface uses four-byte (32-bit) integers for communication between RCAP and the user. The parameters use the native byte-ordering in each network node; they are converted (if necessary) to network byte-ordering before transmission to other RCAP modules on other nodes.

Parameters representing time are expressed in units of 2^{-16} seconds. This format permits the representation of times as short as 15 microseconds and as long as 18 hours (for 32-bit representations). Quantities in this representation are said to be expressed in ‘‘standard temporal units.’’

Parameters expressing probabilities are expressed in ten-thousandths (10^{-4}). Values greater than 10,000/10,000 generate an error. Quantities expressed in this representation are said to be in ‘‘standard probabilistic units.’’

We use a C-like pseudo-code in the interface descriptions and data structure definitions below. By definition, `u_int` refers to a 4-byte unsigned quantity, `u_short` refers to a 2-byte unsigned quantity, and `u_char` refers to a 1-byte unsigned quantity.

4.1.1. Channel Establishment (Sender)

A sending client (C_S) requests a CM channel by invoking the `RcapCmEstablishRequest()` primitive:

```
u_int RcapCmEstablishRequest(
    cmParmblock *parms, u_int *lcid,
    struct in_addr *ipAddr);
```

This primitive implies that all entities involved in supporting a possible CM channel (i.e. the internetwork layer, CM_S, CM_R, and C_R) are contacted. A channel is established only if all entities agree that sufficient resources are available to ensure that the traffic and QOS parameters the client specified for the channel can always be satisfied. The primitive is blocked until an answer is known about the results of the establishment request. In the case of successful establishment, the local channel ID is returned in `lcid` along a return value of `success`. If the establishment fails, the reason for the failure is returned. In addition to the general failure codes listed in [BaM91b], CM channel establishment could generate the following failure codes:

`fail_SendClientBuf`	Maximum buffer size specified by C_S is too small.
`fail_RecClientBuf`	Insufficient buffer space available on receiving host.
`fail_RecClient`	Channel refused by receiving client.

The contents of the data structures are explained below.

Parameter Block

```
typedef struct {
  rcapCmTraffic      *CmTraffic;
  rcapCmRequirements *CmRequirements;
  rcapAddress        *destination;
  rcapUserControl    *control;
} cmParmblock;
```

A `cmParmblock` consists solely of pointers to the other data structures needed in order for channel establishment to take place.

Traffic Characteristics

The sender must specify its traffic characteristics using the following parameters:

```
typedef struct {
  u_int                 STDUmax;
  bool                  CONST_SIZE;
  bool                  CONST_NUM;
  u_int                 T;
  u_int                 Nmax;
  u_int                 Smax;
  u_int                 Savg;
  u_int                 Iavg;
  u_int                 Smin;
  u_int                 S_Sslack;
  struct BufferStruct *Buffer;
} rcapCmTraffic;
```

These parameters are defined in section 2.2. C_S must specify the characteristics of a channel in order to cover the requirements of all future streams which may be mapped onto this channel (cf. section 4.2)

Quality of Service Parameters

The sending client specifies its QOS requirements for the CM channel according to the data structure described below:

```
typedef struct {
  u_int  Dstream;
  u_int  Serr;
  u_int  Werr;
  bool   REPLACE;
  char  *Dummy;
} rcapCmRequirements;
```

These parameters are also defined in section 2.2 above. Addressing and user control information are handled exactly as described in [BaM91b].

4.1.2. Channel Establishment (Receiver)

Channel establishment takes place in three steps at the receiver. First, the client informs the RCAP module that it is willing to accept channel requests on a specified *port number* by invoking the `RcapRegister()` primitive:

```
u_int  RcapRegister(u_short port,
    u_int queue_length)
```

Port numbers are maintained by RCAP and must be globally unique on each end-system. To communicate with a receiver, the sender will have to know the correct port number already, either through prior communication through use of ‘‘well-known’’ port numbers for standard services, or through use of a network name server. This primitive registers the port number with the RCAP module and specifies a maximum queue length for channel requests on the port. It returns one of two values:

`success`	Client registered at requested port.
`err_port_in_use`	Port already in use.

A channel establishment request which arrives for a port that is not registered, or for which the queue is full, will be returned to the sender. Queues and port numbers are released by the `RcapUnregister()` primitive:

```
u_int  RcapUnregister(u_short port,
    u_int queue_length)
```

After registering a port number, the receiving client can obtain requests on the port for CM channels by invoking the `RcapCmReceiveRequest()` primitive:

```
u_int RcapCmReceiveRequest(
  u_short port, cmParmblock *parms,
  rcapAddress *source, u_short *lcid)
```

This primitive is the same as the `RcapReceiveRequest()` primitive described for the message service, except that `parms` refers to the parameters described above for a CM channel. The reader is directed to [BaM91b] for more detailed information.

C_R uses the characterization of the proposed channel, the source address and its own knowledge of available resources to determine whether to accept or reject the request. In particular, C_R looks at the size of the buffer, given by the `Buffer.size` parameter. C_R tries to allocate a buffer that is at least `Buffer.size + S_Rslack` bytes, where `S_Rslack` is chosen by C_R. If a large enough buffer cannot be allocated, C_R rejects the channel request. C_R informs RCAP of its decision via the `RcapCmEstablishReturn()` primitive:

```
u_int  RcapCmEstablishReturn(
    u_short lcid, u_short result,
    u_short reasonCode,
    rcapUserControl *control
    cmParmblock, *parms)
```

This primitive is the same as the `RcapEstablishReturn()` primitive defined in [BaM91b] for RMTP, except that the parameter block is also returned. Since C_R is responsible for allocating the shared buffer on the receiving host, it must return the `Buffer` field to CM to indicate the location of the shared buffer. Rather than returning only part of the parameter block, the entire block is returned.

4.1.3. Channel Teardown

Channel teardown for a CM channel is the same as channel teardown for a message channel, except that additional resources must be reclaimed at the sending and receiving hosts, e.g., the shared buffer. The `RcapCmCloseRequest()` call initiates channel teardown from either the source or the receiver. The primitive is specified as follows:

```
u_int RcapCmCloseRequest(
    u_short lcid,
    u_short reasonCode)
```

Except that it causes additional resources to be reclaimed at both the sending and receiving hosts, this call is the same as the `RcapCloseRequest()` call described in [BaM91b].

4.1.4. Channel Status Request

The `RcapStatusRequest()` primitive provides the performance statistics maintained by the channel. It is described in [BaM91b].

4.2. Service primitives for data transmission

Data transmission corresponds to sequences of *streams*, where a stream consists of the periodic transmission of *stream data units (STDUs)* from C_S to C_R. The traffic and QOS characteristics of each stream may be redefined, subject to certain restrictions: the basic nature of the communication cannot change (e.g., constant-size STDUs, variable number of STDUs per period), and the new characterization cannot exceed the resource demands of the actual channel. The actual channel need not be modified; rather, this mechanism allows the CMTS to possibly optimize its handling of the stream (e.g., allow some buffers to be borrowed by other traffic) and allows for better coordination between C_S and C_R. The following primitives will be used to describe the CM data transmission service:

```
cmOpenStream(u_short lcid,
  cmParmblock *stparms,
  u_int duration,
  u_int d_startup,
  struct userDataBlock *clientData)
cmWaitForOpen(u_short lcid,
  cmParmblock *stparms,
  u_int duration,
  u_int d_startup,
  struct userDataBlock *clientData)
cmWriteSTDU(
  struct StduStruct *stdu)
cmReadSTDU(BufferStruct *buf)
cmCloseStream(u_short lcid)
cmWaitForClose(u_short lcid)
```

C_S invokes `cmOpenStream()` to indicate that a new stream is about to begin on the channel. `lcid` is the local channel ID on which data will be sent. `stparms` are the traffic and QOS parameters as redefined for the stream. As mentioned earlier, only a subset of all channel parameters may be altered and these may only be altered in a restricted manner. Parameters for which the stream parameter may be larger than the value guaranteed by the channel are `T`, `Smin`, and `Dstream`. Parameters for which the stream parameter may be smaller than the value guaranteed by the channel are `Nmax`, `Smax`, `Savg`, and `Werr`.

The `duration` parameter specifies a maximum duration after which the stream will expire as if a `cmCloseStream()` primitive had been invoked. This parameter is passed on to C_R which may use it in allocating resources for the stream, e.g. a file. If a `duration` of 0 is used, the lifetime of the stream is unbounded. On the sender, `d_startup` is the time between the indication of the start of the data stream (i.e. the `cmOpenStream()` invocation) and the start of the first period of the stream on the sender (at time t_0). It is specified by C_S and, as explained in section 2, `d_startup` is *not* included in the overall stream delay `Dstream`. The parameter `clientData` contains a variable-length field which is passed from C_S to C_R without any interpretation by the CMTS. One important example of the kind of data which may be contained in this field is a timestamp to be used in synchronizing two streams. If an illegal value is specified for any of the stream parameters, the `cmOpenStream()` primitive fails.

A receiving client indicates that it is prepared to receive a stream by invoking the `cmWaitForOpen()` primitive, which will block until the next stream is started. Use of the `cmOpenStream()` primitive on the sending host causes the `cmWaitForOpen()` primitive of the corresponding receiving client to unblock, indicating that data transmission is about to begin and communicating the `stparms`, `duration`, `d_startup` parameters and any client data to the receiving client. The value of `d_startup` will have been modified by CM as described in the implementation section. On the receiver, `d_startup` is the time between the return from the `cmWaitForOpen()` call and the start of the stream on the receiving host. Therefore, after returning from the `cmWaitForOpen()` call, C_R must wait for the time indicated by `d_startup` before reading data from the shared buffer (to avoid starvation).

C_S uses the `cmWriteSTDU()` primitive to transfer STDUs from C_S to C_R. The `stdu` parameter is the STDU to be transmitted. Since no explicit interaction between C_S and CM_S is required, this primitive is actually a *pseudo*-primitive, and consists of merely writing data into the shared buffer in time for transmission. The service reserves space for `S_Sslack` extra bytes in the buffer, so that C_S may pre-load that much data in the buffer early, i.e. before the beginning of the corresponding period (time $t=t_{i-1}$ for interval Δt_i). As a result of the invocation of the `cmWriteSTDU()` primitive, STDUs will be retrieved from the buffer and transmitted to C_R before the corresponding period begins on the receiver (i.e. τ_{i-1}, cf. section 2.2), while honoring traffic parameters specified at the beginning of the stream by means of the `cmOpenStream()` primitive.

C_R uses the `cmReadSTDU()` *pseudo*-primitive to retrieve STDUs sent from C_S. The `buf` parameter is a pointer to a buffer in which the STDU is to be placed. Actually, it is the next location in the shared buffer (i.e. the area following the most recent STDU in the buffer). Because these buffers are *pre-allocated*, all data associated with a period should be read by C_R before the end of the corresponding period so that the buffer space can be reused for incoming data. However, extra buffer space is provided so that C_R can fall slightly behind without losing data. As described in section 2.2, C_R can leave up to `S_Rslack` bytes in the buffer after they should have been read (recall that `S_Rslack` is chosen by C_R). The `cmReadSTDU()` primitive requires that STDUs from C_S be put into the shared buffer (by CM_R) before the beginning of the period in which they are to be read.

C_S invokes the `cmCloseStream()` primitive to indicate the end of a stream. This primitive will cause an invocation of the `cmWaitForClose()` primitive to complete at the receiver, indicating the end of the stream to C_R.

4.3. Formalized service definition according to service primitives

We are now prepared to formalize our service definition according to the service primitives defined above. The states for a CM channel are listed in Table 4.1. State transitions are shown in Figure 4.1 for events at the sending host, and in Figure 4.2 for events at the receiving host.

4.3.1. Service as seen by the sending client

Before sending CM data, a client must first request that a CM channel be established to the receiver by invoking the `RcapCmEstablishRequest()` primitive. When this primitive is invoked, the channel will change from the UNDEF state to the SETUP state. If a channel is successfully established, the primitive will return the local channel ID (`lcid`) of the channel, which will be used in all further interactions. Successful channel establishment causes a channel to make a state transition from SETUP to

Table 4.1: Legal states for CM channels

State	Description
UNDEF	Channel is undefined
SETUP	Channel is being established
DELETION	Channel is being torn down
IDLE	Channel is defined, but between streams
PREPARE	Channel is preparing for data transfer
SEND	Channel is currently transmitting a stream of data
RECEIVE	Channel is currently receiving a stream of data

IDLE.

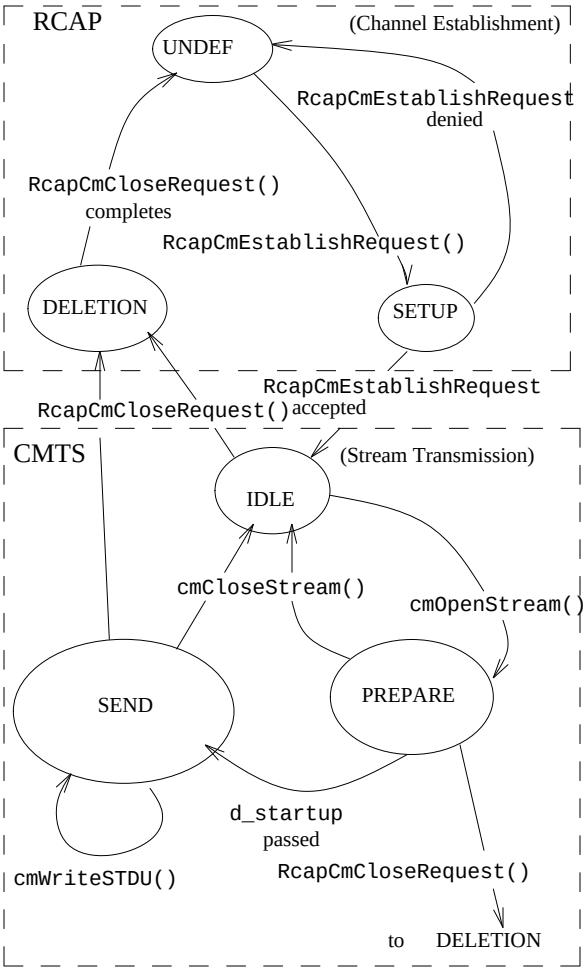


Figure 4.1: Transitions of channel state at the sender

After a channel has been successfully established, data transfer can begin. The sending client (C_S) must signal the beginning of a logical stream before it can begin sending data. The `cmOpenStream()` primitive is invoked for this purpose. When invoking this primitive, C_S has the option of altering some of the traffic and QOS parameters and of specifying additional parameters as described in the preceding section. The `cmOpenStream()` primitive will cause the receiving client's invocation of the `cmWaitForOpen()` primitive to complete, informing the receiving client (C_R) that data transmission is about to begin (see below). `cmOpenStream()` can only be invoked when the channel is in the IDLE state. Successful completion of the primitive causes the state of the channel to change from the IDLE state to the PREPARE state. The stream bit is changed to indicate a new stream (cf. section 5.2).

In the PREPARE state, C_S prepares for transmission by pre-filling the shared buffer. After the time indicated by `d_startup`, the channel changes from the PREPARE state to the SEND state and data transmission can begin.

After the channel enters the SEND state, C_S may send a stream of data to C_R by ''invoking'' the `cmWriteSTDU()` *pseudo*-primitive (i.e. by writing data into the shared buffer). The `cmWriteSTDU()` primitive transmits STDUs from C_S to C_R in such a manner as to honor all QOS and traffic parameters specified for this stream. The channel will remain in the SEND state after completion of this primitive. If the stream is terminated by a channel teardown primitive (i.e. `RcapCmCloseRequest()`), the channel will change to the DELETION state; if the stream is terminated by invocation of a `cmCloseStream()` primitive, the channel will change to the IDLE state.

C_S may invoke the `cmCloseStream()` primitive for any channel in the SEND or PREPARE states. Completion of this primitive will cause the channel to change from its current state to the IDLE state and will cause the corresponding `cmWaitForClose()` to complete at the receiver.

When the sending client is finished with the channel, it can be torn down by invoking the `RcapCmCloseRequest()` primitive. Invocation of this primitive causes the channel state to change from any state (besides UNDEF) to the DELETION state. When the primitive completes, the channel changes from the DELETION state to UNDEF. The channel could also be torn down by the network or receiving client, in which case the

sending client would be informed that the channel no longer exists.

4.3.2. Service as seen by the receiving client

Before a client can receive data, a channel must be established to it from a sender. First, the receiving client C_R informs the RCAP module that it is willing to receive channel establishment requests on a specified port via the `RcapRegister()` primitive. After that, C_R may receive requests for channels by invoking the `RcapCmReceiveRequest()` primitive. When a channel request is received, the channel enters the SETUP state, matching the state at the sender (see above). C_R then decides whether it is willing to accept the request and indicates its decision using the `RcapCmEstablishReturn()` primitive. If C_R denies the request, the sender's establishment request fails and the state of the channel changes from SETUP to DELETION at both sender and receiver. If C_R accepts the channel, it is established, causing the state of the channel to change from SETUP to IDLE at both the sender and the receiver.

After a channel has been successfully established, data transfer can begin. Upon entering the IDLE state, C_R immediately invokes the `cmWaitForOpen()` primitive, which will complete when a new stream is begun at the sender via the `cmOpenStream()` primitive. Completion of the `cmWaitForOpen()` primitive causes the traffic and QOS parameters, the `duration` value, and the `d_startup` value for the new stream to be delivered to C_R along with the `clientData` field, and causes a state transition from IDLE to PREPARE.

In the PREPARE state, C_R prepares to receive STDUs. When this state is entered, C_R immediately invokes the `cmWaitForClose()` primitive. This primitive will return only when the stream has been closed. After the time indicated by `d_startup`, the channel changes from the PREPARE state to the RECEIVE state and C_R is allowed to begin reading data.

After the channel enters the RECEIVE state, C_R may receive STDUs from C_S by invoking the `cmReadSTDU()` *pseudo*-primitive. Actually, ''invocation'' of this primitive consists of reading from the shared buffer and updating the descriptors. The channel will remain in the RECEIVE state after completion of this primitive. The stream may be terminated by a `cmCloseStream()` invocation at the sender, which will

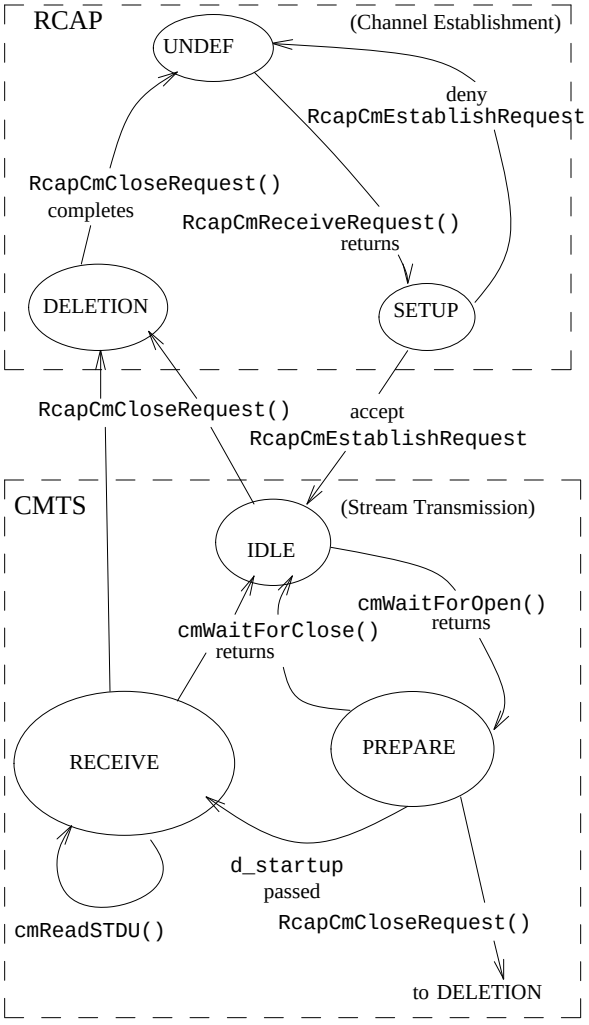


Figure 4.2: Transitions of channel state at the receiver

cause the call to `cmWaitForClose()` to return; or by a `RcapCmCloseRequest()` by either the source, the destination, or the network, as described above for `cmWriteSTDU()`.

The receiving client may tear down a channel by invoking the `RcapCmCloseRequest()` primitive. Initial invocation causes channel state to change from any state (other than UNDEF) to DELETION. After the tear down primitive completes, the channel state is changed to UNDEF. The channel could also be torn down by the sending client or the network, in which case the receiving client would be informed that its channel no longer exists.

4.4. Timing of Stream at Receiver

The processing of data by the receiving client must be synchronized in some way with data production by the sending client. Several possible

mechanisms could be used for communicating timing information between C_S and C_R. In this section we would like to analyze the suitability of CMTS for each of the four mechanisms of which we are aware.

First we define

$r(\Delta\tau_i) \triangleq$ Function determining timing of STDU consumption at the receiver.

Table 4.2: Possible definitions for $r(\Delta\tau_i)$

I.	Known *a priori* for the duration of the stream, e.g. uncompressed voice $\rightarrow$ 1 byte per 125 μsec for each voice channel in PCM coding.
II.	Direct function or consequence of the decoding process at the receiver, e.g., compressed video transferred as a byte stream in which the frame boundaries embedded in the compressed stream.
III.	STDUs contain explicit ''time-stamps'' added by the sending client. These are passed directly on to the receiving client as part of the CM data. This kind of timing information is particularly relevant for stored streams.
IV.	Timing is generated by arrival process of STDUs at the sender ($\lambda[\Delta t_i]$), which must be reconstructed at the receiver. This case includes those receiving applications which do not perform any timing functions themselves, but rely on the arrival of STDUs for timing.

I-III above are directly supported by CMTS. We chose not to support option IV directly in the CMTS layer because such a service would only be required by a fraction of CM clients and can be implemented as a local service on top of CMTS using option III with timers to schedule when to ''deliver'' the next STDU to the client.

5. Functional Description of CM layer and Specification of CMTP protocol

This section describes the functionality of the CM layer and the *Continuous Media Transport Protocol (CMTP)* which facilitates communication between CMTS peers to provide this functionality.

5.1. Functionality of CMTS layer

The functionality of the CM layer consists of the functions performed by the CM entities at the sending (CM_S) and receiving (CM_R) end-systems.

Sending host (CM_S)

The CMTS entity on the sending side (CM_S) must perform functions for connection establishment and teardown and for data transport.

Channel establishment (handled by RCAP on behalf of CM_S):

- Translate parameters from CM to network characterization
- Acquire resources on sending end-system for CM channel
- Acquire a suitable real-time network channel
- Forward channel request to peer on receiving host (CM_R)
- Set up data structures required for data transport

Channel teardown (handled by RCAP on behalf of CM_S):

- Release resources and network channel

Data Transport (sending):

- Notify peer on receiving host (CM_R) of start of a new stream (*)
- Get data from sending client
- Packetize data
- Schedule transmission of data units and send to peer (*)
- Notify peer on receiving host (CM_R) of end of stream (*)

Receiving host (CM_R)

Channel establishment (handled by RCAP on behalf of CM_R):

- Perform tests to determine whether requested channel can be serviced
- Pass on request to receiving client (C_R)
- Acquire resources on receiving end-system for CM channel
- Set up data structures required for data transport

Channel teardown (handled by RCAP on behalf of CM_S):

- Release resources and network channel
- Notify client (C_R)

Data Transport (receiving):

- Notify receiving client (C_R) of start of a new stream
- Verify packet sequence number to detect lost data
- Verify checksum to detect corrupted data
- If data is corrupted or lost and replace option has been specified, replace each missing packet with *Dummy*
- Unpacketize data and put in shared buffer
- Update descriptor with condition of data (valid, corrupt, lost)
- Signal client of receipt of data if client is waiting for data
- Notify receiving client of end of stream

To perform those tasks marked with a (*), a protocol must be defined to allow CM_S to communicate with CM_R.

5.2. Continuous Media Transport Protocol (CMTP)

The first version of the CMTP protocol could be kept relatively simple. This simplicity results primarily from the fact that several of the communication functions needed in conventional data communication (in particular, retransmissions for error correction, flow control, etc.) are not required in order to provide the CMTS service. Regarding retransmissions, we take the position (stated in [FeV90]) that most real-time applications will not be able to wait for retransmissions, and even if they could, the amount of data which would need to be stored to perform retransmissions on a high bandwidth-delay product network could not be justified for CM clients, which do not require perfectly reliable service. Similarly, resetting a data stream to an earlier status (period) is not possible as the resource requirements needed to set check-points in general are prohibitive for storing an intermediate status of a stream. Therefore in the case of a serious protocol error, channel tear-down and establishment of a new channel (with a new stream) seem to be the most appropriate measures.

The simplex nature of the real-time connections used also constrains the potential dialog between sender and receiver within such a connection, allowing a simpler protocol to suffice. In the design of the CMTP protocol, we have assumed that a data stream between C_S and C_R is transmitted via exactly one (uni-directional) connection between C_S and C_R. This connection thus represents a data connection. As no multiplexing takes place in the CM transport layer, a one-to-one mapping of the addresses of communicating clients to the address of the network connection (connection-id provided by RCAP in the case of XUNET II) can be used to solve the addressing problem.

As an extension to the current design, we assume that CM_S and CM_R are able to (reliably) exchange control information concerning the state of the data connections presently established between them. In a similar way, we assume that the CMTS service reliably transfers client-control information between C_S and C_R to support an application-defined protocol between them (separate communication, in addition to the exchange of a data stream). This solution can be viewed as an "out-of-band-signaling" between the communicating CM clients.

Until now, the CMTP protocol has only been specified for communication within the data connection. Experiences of the CMTS implementation are considered to be indispensable prior to a protocol extension and will be taken into account in the completion of the CMTP protocol.

Specification of a communication protocol must cover three fundamental aspects (cf. [Wol81]):

- *syntax* of Protocol Data Units (PDUs) exchanged, i.e. data formats
- *semantics* of PDUs, i.e. their meaning for the communicating entities
- possible sequences of PDUs exchanged over time *(timing)*

The CMTP protocol is based on the use of three types of protocol data units: ON_PDU, OFF_PDU, and DATA_PDU. Each PDU is carried in exactly one network packet, i.e. PDUs are not fragmented. Because PDUs are not fragmented, the `len` parameter returned by the underlying internetwork protocol (cf. section 3.1) can be used to indicate the length of variable-size packets. Therefore, no length field is transported by CMTP, even though CMTP PDUs may be of

variable length. The syntax of the three types of PDUs is shown in Figure 5.1 and their semantics are described below.

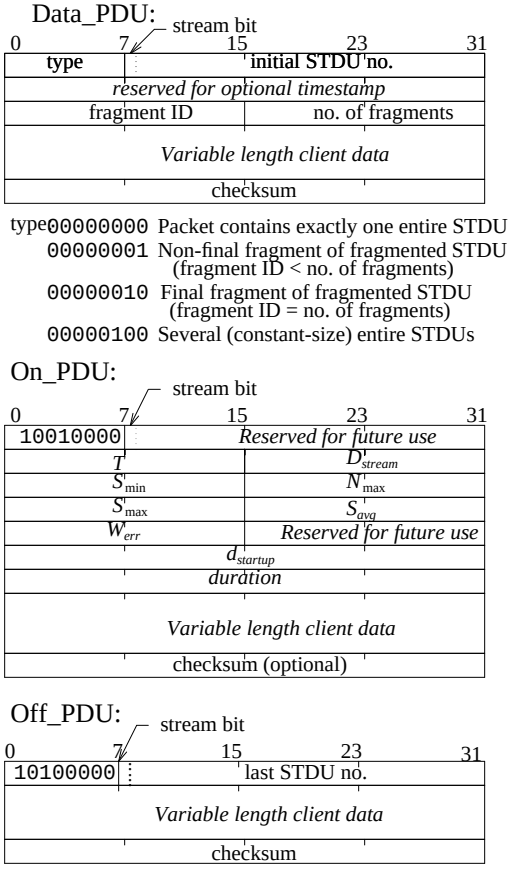


Figure 5.1: Format of CMTP PDUs

The ON_PDU signals CM_R that a new stream is about to begin on the channel. Each stream is identified by an alternating *stream bit*, which ensures that, even if an OFF_PDU and both subsequent ON_PDUs are lost, a new stream will be noted because of the change in the stream bit. The stream bit is therefore carried in each PDU. In DATA_PDUs and OFF_PDUs, the stream bit is the most significant bit in the STDU number field (see Figure 5.1). Additionally, the ON_PDU contains those traffic and QOS parameters whose values are allowed to be modified for the duration of the stream. As described in section 2, each parameter can only be varied in one direction with respect to the channel characteristics. This restriction simplifies validity checking and ensures that changing one parameter will not invalidate a parameter which has not been changed.

The ON_PDU also contains two additional parameters, $d_{startup}$ and *duration*, which are not parameters of the channel, but are specified by the client at the beginning of each stream as described in the previous section. $d_{startup}$ is the amount of time between the `cmOpenStream()` call made by C_S[7] and the beginning of the stream at the C_S/CM_S interface. This time is used by C_S to pre-fill the buffer at the sender. $d_{startup}$ is transferred to the receiver, because C_R must wait at least as long as $d_{startup}$ after the receipt of the OPEN_PDU by CM_R before it is allowed to start reading data. Of course $d_{startup}$ must be included in the end-to-end delay seen by the application, but, as described in section 2, this delay is outside the realm of the CM service because it occurs before the beginning of the stream (on both the sender and receiver); therefore, it is not included in the end-to-end stream delay, D_{stream}.

If the ON_PDU is lost, these stream parameters are unknown at the receiver. Therefore, the ON_PDU is sent twice during the startup time (given by $d_{startup}$).[8] In the unlikely occurrence that both ON_PDUs are lost, the only solution in the current design is for CM_R to tear down the channel. Future work will add a full duplex control channel that can be used to recover from a lost ON_PDU more gracefully. The control connection is described in more detail at the end of this section.

The sending client may optionally specify the *duration* of the stream. The CM layer notes this parameter and passes it on to the receiving client, which may use it for resource allocation (e.g. files) or its own timing. The *client data* field is for control data that C_S may wish to pass on to C_R, such as a timestamp to allow the beginning of two streams arriving from the same host to be synchronized. An optional checksum protects the entire packet.

The OFF_PDU is used to signify the end of the stream. It specifies the number of the last STDU in the stream, so that lost STDUs can be detected at the end of the stream.

The DATA_PDU carries data from the shared buffer. Figure 5.2 depicts the mapping of STDUs into DATA_PDUs (which also specifies the mapping into packets, as each PDU is transported in exactly one packet). Depending on the type of

[7] Where the ''time of the call'' is defined to be the time when the client's call *returns.*

[8] Of course the value of $d_{startup}$ and *duration* must be modified appropriately in the second ON_PDU.

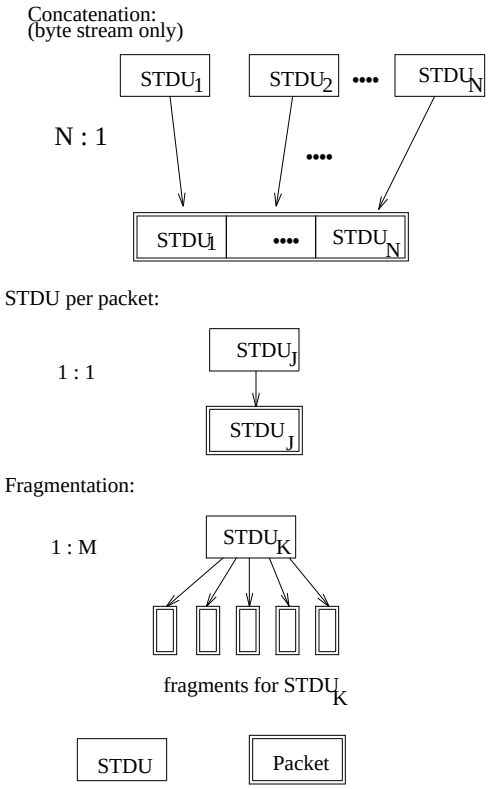


Figure 5.2: Mapping of STDUs into packets (DATA_PDUs)

stream, a single DATA_PDU may contain one STDU, a fragment of an STDU, or multiple STDUs (only if the stream is a byte stream). The *type* field indicates the exact nature of the data in the PDU. The *initial STDU number* indicates the number of the first STDU contained in the packet. This is also the only STDU in the packet unless the channel is carrying a byte stream. For a stream that is not a byte stream, the *no. of fragments* field indicates the number of fragments which comprise the current STDU and the *fragment ID* indicates the fragment number of the enclosed fragment within the current STDU. For a byte stream, the *no. of fragments* field indicates the number of bytes in the packet and the *fragment ID* field is unused. The optional *timestamp* is used by CM for debugging and to collect performance statistics. In the current design, neither this timestamp nor the statistics are available to the client.

Figure 5.3 shows a typical sequence of events on a CM channel. It maps the actions performed by entities to each type of PDU. This figure shows two successive streams, s_0 and s_1, on the same channel. A `cmOpenStream()` call by

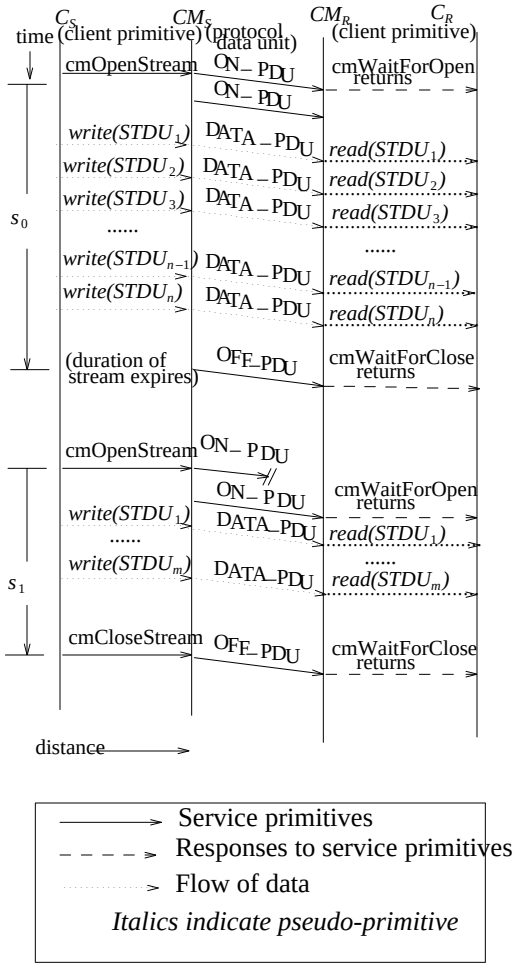


Figure 5.3 Sequence of events for a typical stream

the client marks the beginning of s_0, and, causes CM_S to send an ON_PDU to CM_R, which then unblocks the call C_R made to `cmWaitForOpen()`. After $d_{startup}$ has passed, CM_S periodically transfers data written into the shared buffer by C_S to CM_R via DATA_PDUs. Upon receipt of this data, CM_R puts it into the buffer it shares with C_R. A `cmCloseStream()` call by C_S or expiration of the *duration* of the stream, causes CM_S to send an OFF_PDU to CM_R, which then unblocks the call C_R made to `cmWaitForClose()`, indicating the end of the stream to C_R.

A more formalized description of the protocol, including its basic timing, is given by the state diagrams at the sender and receiver shown in Figures 5.4 and 5.5 respectively. These diagrams are similar to the state diagrams shown in the previous section, except that they have been extended to include sending of protocol data units on the sender and the effect of receiving

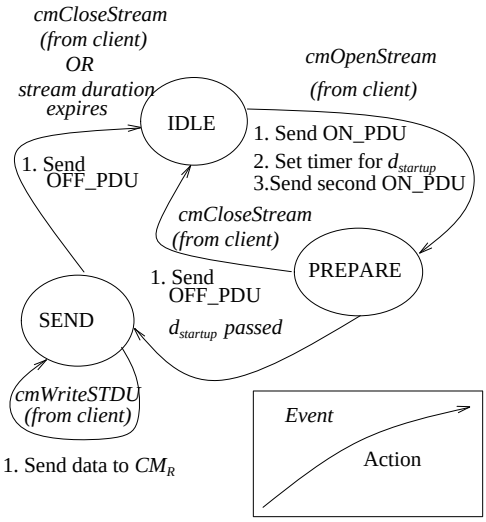


Figure 5.4: Channel state transitions at CM_S (data connection only)

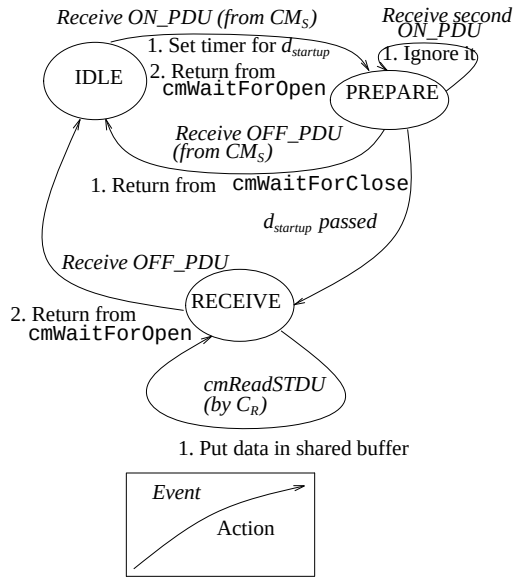


Figure 5.5: Channel state transitions at CM_R

PDUs on the receiver, as well as some other basic actions of CM_S and CM_R. It should be noted that these diagrams are incomplete since the states and actions required to recover from protocol errors has been omitted for the sake of clarity. A refinement of the actions actually taken by these CM entities is given in section 6.2. States and transitions relating to channel establishment and teardown have been omitted. As in section 4, a channel is in the IDLE state when it is between streams, in the PREPARE state during the startup delay ($d_{startup}$) and in the SEND or RECEIVE state when sending or receiving CM data respectively.

As mentioned above, we propose a future extension of the design which includes a control connection. Such a connection would be a low-throughput, duplex connection offering minimal or no performance guarantees, except reliable transfer. Two types of control messages would be carried on the connection:

1. User-control information which the clients (C_S, C_R) could use to implement their own bi-directional protocol
2. CM-CM control information, which could consist of the following possibilities (details to be determined through experimentation with the prototype).

    - ACK/NAK of ON_PDU. Acknowledgement of ON_PDUs would eliminate uncertainty at the sender as to the state at the receiver. Negative acknowledgement of reception of ON_PDUs (CM_R has noticed a new stream either because of reception of an OFF_PDU or new stream bit, but didn't receive the ON_PDU) allow the sender to stop sending the current stream and start again.
    - ACK of OFF_PDU to eliminate uncertainty.
    - Indication of buffer overflow at the receiving interface.
    - PING to check if sender is still alive when no data is has been received for a given interval and no OFF_PDU has been received.
    - Other possible error situations discovered through experimentation with the prototype implementation.

6. Implementation Considerations

This section will describe a framework for an implementation of the continuous media transport service. Although the service could be implemented differently, it was designed with this implementation in mind. We first list some of the basic problems any implementation of a data transport service must solve. We then describe a proposed implementation for solving these prob-

lems, taking into account the periodic nature of CM to eliminate synchronous interactions between the client and the service provider. We would like to emphasize at this time that we want to provide a basic service suitable for many different kinds of CM clients. Although most clients will want to use the features of CMTS directly, we expect that thin layers of library routines would be implemented to provide an interface tailored to the needs of less sophisticated clients. At the end of this section, we will sketch a few examples of these layers.

One of the fundamental goals of the realization of the CM Transport Service has been to use buffers in the source and destination nodes to smooth variations both in the arrival process of data to be transmitted and in network delays. The motivation behind this decision is the expectation that buffers will be relatively inexpensive compared with the total network resources (including throughput, buffers, scheduling, etc.) conserved by smoothing the data traffic.

6.1. Problems to be solved in the implementation

Inherent to any transport service are a number of problems which have to be solved in the implementation. In the description that follows, the problems are described in terms of *producers* and *consumers* of data *at the interface between clients and the service.* Therefore, a producer refers to either C_S or CM_R. It then follows that in this description, C_R is a consumer of data ''produced'' by CM_R (and not by C_S). The following are some of the most important problems to be solved:

(P1) How does a producer of data tell a consumer where to find it?

This problem is referred to as the *data location transfer* problem in [GoA91]. The location of data to be consumed can be transferred from producer to consumer either via an explicit interaction on each data transfer or via shared state, or it can be implied by an earlier agreement.

(P2) How is data transferred from producer to consumer?

This problem is referred to as the *data transfer* problem in [GoA91]. It requires either physical copying or VM remapping.

(P3) How are producers and consumers synchronized with each other?

As stated above, we are referring to producers and consumers across the client/service interface, not between sending and receiving clients. Therefore, we must be sure that the producer does not produce data too quickly, in which case some might be dropped due to buffer overflow; and the consumer must not try to consume invalid data (i.e. data which has yet to be produced). The problem of producer/consumer synchronization in accessing the shared buffer is referred to as the *synchronization* problem in [GoA91].[9] The synchronization problem manages the availability of either data or empty buffer space (i.e. with the implementation of the `cmReadSTDU()` and `cmWriteSTDU()` pseudo-primitives) as well as the beginning and end of streams (i.e. implementation of the `cmOpenStream()`, `cmWaitForOpen()`, `cmCloseStream()` and `cmWaitForClose()` primitives).

(P4) How is the time for data transmission chosen?

This is essentially the *control* problem of [GoA91], which deals with the issue of I/O initiation. This problem cannot be isolated from the actual network transmission, as transfer of data across the sending-client/service interface results in a corresponding data transmission on the network, which will then cause a data transfer across the service/receiving-client interface. For standard event-driven services, transfer of data across the client/service interface at the sending host occurs when data availability is explicitly signalled by the sending client (i.e. via a *send* call). Then, the time for network transmission is chosen to be the next available time as implemented via queueing at the network interface. At the receiving host, transfer of data across the service/client interface occurs when data has been received from the network and the receiving client has requested data via a synchronous request (i.e. *receive*). Since we wish to eliminate synchronous client/server interactions, data transfer across the sending client/service interface must be arranged in advance by some kind of agreement between the producer and the consumer.

[9] The synchronization and control problems are described as one combined problem (the *synchronization and I/O initiation* problem in [GoA91].

Then each can act according to the agreement, without requiring synchronous interactions. Solutions to satisfy these requirements will need operating systems support.

(P5) How are errors in the data indicated to the receiving client?

Some mechanism must exist for communicating the location of data errors and the type of error (i.e. data not delivered vs. data corrupted).

6.2. Framework for a proposed implementation

The framework for the proposed implementation will be described in three parts: (1) the general structure of the solution; (2) setup and teardown of a connection for transferring CM data; (3) steps involved in the actual transfer of data. Throughout this section we will refer to the problems (P1, ... , P5) described in the previous section.

6.2.1. General structure of the solution

To allow for data transfer between producers and consumers without explicit interactions on each transfer, the location of the data to be transferred (P1) must be known ahead of time by both parties, i.e. implied due to an earlier agreement. To solve this problem and to minimize physical copying in data transfer (P2), a shared circular buffer is used for transferring data between producers and consumers. Figure 6.1 gives a conceptual illustration of the shared buffer at the sending-client/service interface (B_S).

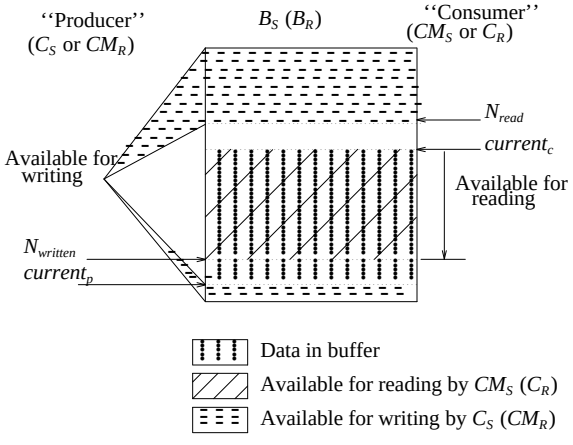


Figure 6.1: Conceptual view of shared circular buffer

Synchronization between sender and receiver can take place using the two shared synchronization variables N_{read} and $N_{written}$ (P3). N_{read} is updated by the consumer to indicate the number of bytes read from the buffer, and $N_{written}$ is updated by the producer to indicate the number of bytes written into the buffer. Each entity keeps its own pointer into the buffer to indicate its current location ($current_c$ for the consumer and $current_p$ for the producer). For variable-size STDUs, (and optionally for constant-size STDUs) STDU boundaries are indicated by descriptors (one per STDU). At the sender, the descriptors include a pointer to the start of each STDU in the buffer, a length field, and a flag to indicate whether the data is present or absent. At the receiver, the flag is extended to indicate error conditions (P5), as well as the presence/absence of data. The structure of the buffer shared at the receiver (B_R) is shown in Figure 6.2. Of course, these descriptors must reside in memory shared by the service and the client.

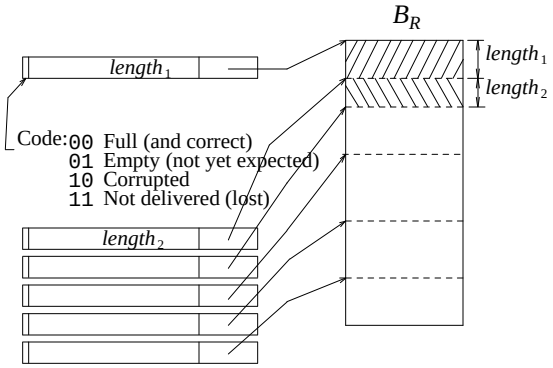


Figure 6.2: Shared buffer with descriptors (receiver shown)

For channels transferring constant-size STDUs with a size of 1 byte (byte stream), an optimization is made by allowing each descriptor to cover a range of STDUs with identical flag values. In this case, the length field indicates the length of the range in bytes (or STDUs).

In order to eliminate synchronous producer/consumer interactions during data transmission, we use the knowledge we have about the CM stream to schedule the time at which data is transferred between C_S and CM_S, rather than requiring an explicit interaction (P4), i.e. CM_S checks the shared buffer for data at the beginning of each period. Network transmission of all packets to be sent during that period is scheduled at that time. When data arrives on a channel, CM_R stores it immediately into the shared buffer, and C_R is free to get the data whenever it wants, as long as it meets the obligations

specified in earlier sections. In this implementation, we make no assumption about clocks being synchronized, but only assume that the relative drift between clock rates of different machines is negligible.

6.2.2. Connection establishment and teardown

Connection establishment consists of three steps: (1) translating CM-layer parameters into network-layer parameters; (2) reserving end-system resources (buffers and CPU time) for CM-layer activities; and (3) acquiring a network-layer channel with the required parameters.

6.2.2.1. Translation of parameters

The parameters for the network-layer service described in section 3 (s_{max}, $x_{\min}$, x_{ave}, *I*, *D*, *Z*, *J*, *U*, and *W*) must be derived from the parameters specified for the CM-layer service described in section 2.2 ($STDU_{max}$, *CONST_LEN*, *CONST_NUM*, *T*, N_{max}, S_{max}, S_{avg}, I_{avg}, $S_{\min}$, S_{Sslack}, D_{stream}, S_{err}, W_{err}, *REPLACE*, *Dummy*, and *Buffer*). For simplicity, the prototype will use only the network-layer service which provides deterministic delay bounds without controlling delay-jitter.[10]

In translating parameters from the CMTS model to the RTIP model, a decision must be made as to how the burstiness of the CM stream will be managed. One possibility is to simply reserve resources based on some aggregate data rate and then accept some loss during bursts of high data rate. This solution, however, will be unacceptable to a large range of clients who do not wish the quality of service provided to decrease during bursts of high data rates. There are three possibilities for reducing the probability of data loss due to burstiness: (1) Smooth out the burst by spreading its transmission over a number of periods (thus incurring additional delay and buffer requirements at the sender); (2) reserve extra network throughput to be used to transmit bursts (expensive if network throughput is a scarce resource); (3) adapt the coding process to network conditions. (Network conditions can cause the shared buffer at the sender to become full or empty. On detecting one of these states, the sender may be able to adjust its coding algorithm to decrease or increase its data rate. Likewise, a receiver may be able to adjust its rate of consumption of data, when it notices the buffer it shares with CM_R is becoming full or empty.)

As mentioned above, the method used to reduce data loss for a bursty stream may affect the end-to-end delay of the stream. This fact points out the general problem we have in partitioning the end-to-end delay bound, D_{stream}, among its component delays as specified in the equation below,

$$D_{stream} = d_{nmax} + d_{sm} + d_{j,} \qquad \text{(i)}$$

where d_{nmax} is an upper bound on the total time spent in the network, consisting of propagation delay, service times in the nodes (including transmission delays), and queueing delays; d_{sm} is the delay allowed for smoothing the traffic pattern at the sender; and d_j is the delay introduced in compensating for delay jitter, where delay jitter is defined as the difference between the maximum delay and minimum delay experienced by packets on the channel. The maximum value of delay jitter is therefore bounded by $d_{nmax} - d_{nmin}$, where d_{nmin} is a lower bound on the total time spent in the network, consisting of only the minimum propagation delay and service times.

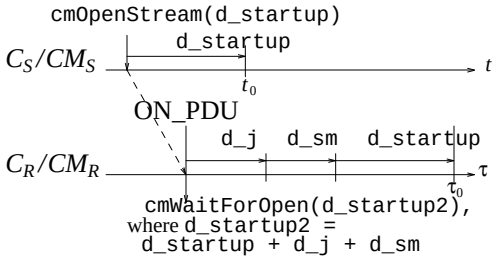


Figure 6.3: Timing diagram of start of stream

If an OPEN_PDU would travel with delay d_{nmin}, a later data packet with a delay greater than d_{nmin} would cause short-term starvation in the stream, resulting in a gap in the stream as seen by C_R, even if the network delay of the packet is less than the delay bound, d_{nmax}. However, if CM_R delays the start of the stream at the receiver by an additional time equal to $d_{nmax} - d_{nmin}$, C_R will never starve due to delay jitter (See Figure 6.3). In other words, we must reduce the total delay available, D_{stream}, by an amount d_j, which can then be used to compensate for delay jitter as just described.[11] Therefore,

---

[10] Simplicity only in avoiding the calculation of *J*, the requested delay-jitter bound (i.e. in deciding how to partition resources between buffering at the receiver and buffering, delay, and delay-jitter bounds in the network). Other than that, incorporating the jitter bound requires no extra work or complexity, as described in the following paragraphs.

[11] We should mention that if the clocks on the two end systems are synchronized to within some given

$$d_j = d_{nmax} - d_{nmin}.$$

If the network provides a bound on delay jitter, d_{nmin} is higher, and hence d_j becomes lower, leaving more of the delay allocation for the network and conserving buffer space at the receiver.

The preceding paragraphs describe the kind of tradeoffs which exist and have to be solved when partitioning the end-to-end delay into its component delays. Several criteria should be considered in deciding on this partitioning for a given channel: D_{stream}, the maximum end-to-end delay for the channel; B_S and B_R, the amounts of buffer space available in the sending and receiving end-systems respectively; the expected delay-jitter in the end-systems themselves; and the relative ''cost'' of network throughput, delay-guarantees, delay-jitter guarantees, and buffer reservations within the network. In particular, CMTS may decide to smooth the data stream of a client in order to decrease its throughput requirements; thereby requiring tighter delay and (possibly) delay-jitter bounds from the network than if the traffic were not smoothed (because of the delay incurred for smoothing). On the other hand, an application with tight delay bounds (i.e. close to the bound provided by the network alone) or which is operating on an end-system with limited buffer space available, may not be able to afford smoothing.

It is clear that the tradeoffs involved are complex and some kind of cost model must be developed in order to intelligently partition the end-to-end delay. Future work will address this problem. To simplify the initial design and implementation, we will assume $d_{nmin} = 0$ and try to allocate delay evenly between network delay, delay-jitter compensation, and smoothing (i.e. we will choose $d_j = d_{nmax}$ and define the proportionality constants $k_{nmax} = k_{sm} = k_j = 1$). The *smoothing interval*, I_{sm}, is defined to be the number of periods over which smoothing of data transmission occurs, i.e. all data presented for transport within the interval Δt_i (= $[t_{i-1}, t_i)$), will be sent before $t_i + (I_{sm} \times T)$. However, I_{sm} is an integer (signifying an integral number of periods), and since there is no point in making I_{sm} larger than the averaging interval I_{avg}, I_{sm} is calculated as follows,

$$I_{sm} = \min\left\{ I_{avg}, \left\lfloor \frac{D_{stream}}{(k_{nmax}+k_{sm}+k_j)T} \right\rfloor \right\}$$

$$= \min\left\{ I_{avg}, \left\lfloor \frac{D_{stream}}{3T} \right\rfloor \right\},$$

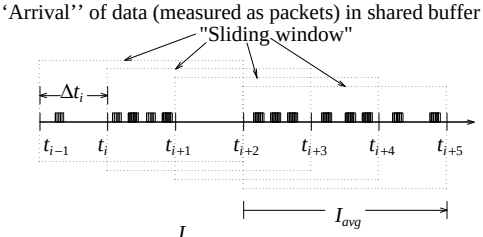
$$d_{sm} = T \times I_{sm}$$

and (as a consequence of the assumption that $d_{nmin} = 0$ and that $k_{nmax} = k_j = 1$),

$$d_{nmax} = d_j = \frac{D_{stream} - d_{sm}}{2k_{nmax}}$$

$$= \frac{D_{stream} - d_{sm}}{2} \qquad \text{(ii)}$$

The traffic pattern seen by lower levels will depend on the rate control mechanisms implemented in the CM layer. Leaky bucket smoothing and rate control will be implemented using a credit scheme for packets. A sample of the credit scheme described below is given in Figure 6.4.



$$I_{avg} = \frac{I}{T} = 3$$

$$decr_{\min} = 1$$

$$decr_j = max(decr_{\min}, n_j)$$

$$incr_j = decr_{(j-Iavg+1)}$$

$$credits_j = credits_{j-1} - decr_j + incr_j$$

Time t_j	Packets in buffer at t_{j-1}	n_j	$decr_j$	$incr_j$	$credits_j$
t_i	0+1=1	1	1	---	---
t_{i+1}	0+4=4	4	4	---	2
t_{i+2}	0+0=0	0	1	1	2
t_{i+3}	0+3=3	2	2	4	4
t_{i+4}	1+3=4	4	4	1	1
t_{i+5}	0+2=2	1	1	2	2

Figure 6.4: Implementation of rate control scheme using credits

To send a packet, a channel must spend one packet credit from its total, which is indicated by the state variable *credits*. Packets can only be sent as long as $credits > 0$. Since $S_{\min}$ is defined as the minimum number of bytes *expected* to be transmitted in each period, *credits* is decremented by at least $decr_{\min}$ each period, where

---

amount of time t_{synch}, then the maximum uncertainty in the delay of the OPEN_PDU is t_{synch}, and hence the component of D_{stream} due to delay jitter is also equal to t_{synch}.

$$decr_{\min} = f(S_{\min}),$$

even if less data is actually sent. Each period, *credits* is incremented by an amount $incr_i$, which is calculated to ensure that the client does not exceed its average rate characterization specified using S_{avg} and I_{avg}. Recall that the average rate characterization must be valid for any observation interval of length I_{avg} which starts at the beginning of a period. In other words, the characterization of average rate defines a ''sliding window'' of length I_{avg}, which ''slides to the right'' (i.e. in the direction of increasing time) one period at a time. At no time may the amount of data contained in this sliding window exceed $I_{avg} \times S_{avg}$ bytes. Similarly, the number of packets constructed by the CM layer cannot exceed I / x_{ave} packets within a sliding window of length I. Since there is no point in having two different averaging intervals, we will choose $I = T \times I_{avg}$, and the two sliding windows are identical. The sliding window is shown in several positions in Figure 6.4.

If the number of credits in the first position of the sliding window is valid, we can guarantee it remains valid in all succeeding positions by incrementing *credits* by the amount of data that ''falls out'' of the left hand side of the window when it moves to the right. In other words, if $credits_0$ is ''valid'', and

$$credits_i = credits_{i-1} - n_i + n_{(i-Iavg+1)},$$

(where $credits_i$ is defined to be the number of credits remaining after the interval Δt_i, i.e. at time t_i, and n_i is defined to be the number of packets sent in interval Δt_i) then the following equation always holds at time t_i,

$$credits_i + \sum_{j=i-Iavg+2}^{i} n_j \le I_{avg} \times S_{avg.}$$

The inequality can be made an equality if we increment *credits* by the same amount as it had been decremented at the end of the interval which is now falling out of the window.[12] So, if $credits_0$ is ''valid'', and

$$credits_i = credits_{i-1} - decr_i + decr_{(i-Iavg+1),}$$

where $decr_i = min(decr_{min,}\ n_i)$, then

$$credits_i + \sum_{j=i-Iavg+2}^{i} n_j = I_{avg} \times S_{avg.}$$

Therefore, each period, *credits* is incremented by the exact amount it had been decremented at the start of the period which is now falling out of the window. In other words,

$$incr_i = decr_{(i-Iavg+1),} \qquad \text{(iii)}$$

where the interval $\Delta t_{(i-Iavg+1)}$ is falling out of the sliding window after the interval Δt_i has passed. To completely specify the credits scheme, we must determine the minimum decrement per period ($decr_{\min}$), the initial value of *credits* ($credits_0$), and initial values for $incr_i$ (i.e. for $incr_1, incr_2, ..., incr_{(Iavg-1)}$).

To ensure that all data is sent with a smoothing delay less than or equal to d_{sm}, it is sufficient to ensure that all data presented by C_S during any interval of length d_{sm} (= $I_{sm} \times T$) is sent by the end of the interval. We can use the average rate parameters (I_{avg}, S_{avg}, and $S_{\min}$) to calculate the maximum number of bytes which could be presented for transmission during a smoothing interval. See Figure 6.5 for the relationship between I_{avg} and I_{sm}.

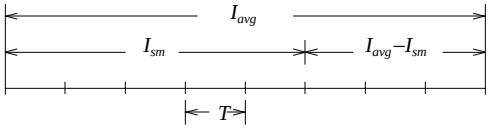


Figure 6.5: Relationship between I_{sm} and I_{avg}

The maximum number of bytes which could correspond to a smoothing interval is calculated by subtracting the minimum amount of data which can be expected within I_{avg}, but not within I_{sm} (i.e. $S_{\min} \times (I_{avg} - I_{sm})$) from the maximum amount of data which may be presented in I_{avg} (i.e. $S_{avg} \times I_{avg}$). Therefore, the amount of data which could be required to be sent within any smoothing interval is given by

$$S_{avg} \times I_{avg} - S_{\min} \times (I_{avg} - I_{sm}).$$

From this formula, we see that the smoothing delay bound can be met if at least S_{trans} bytes are sent per period, where

$$S_{trans} = \min\left(\frac{S_{avg} \times I_{avg} - S_{\min} \times (I_{avg} - I_{sm})}{I_{sm}}, S_{max} \right)$$

Note that if $I_{sm} = I_{avg}$, then $S_{trans} = S_{avg}$. We can now calculate the parameters for the network-layer service.

[12] Because $decr_i = min(decr_{min,}\ n_i)$, $decr_i \ge n_i$.

(a) Traffic and performance parameters for byte stream ($STDU_{max}$ = 1)

The maximum packet size, s_{max} is limited by the maximum packet size supported by the network layer, which we will call s_{net}, as well as by the user-specified error control size, S_{err}, so that

$$s_{max} = \min(s_{net}, S_{err}) \quad (1a)$$

$x_{\min}$ depends on the largest number of packets, n_{trans}, which may need to be transmitted during a period in order to keep the smoothing delay under d_{sm}.

$$n_{trans} = \left\lceil \frac{S_{trans}}{s_{max}} \right\rceil \quad (2a)$$

$$x_{\min} = \frac{T}{n_{trans}} \quad (3a)$$

As previously stated, the averaging interval I is the same as I_{avg} except that I is expressed in terms of units of time and I_{avg} is expressed in periods. To calculate x_{ave}, we first calculate n_{avg},[13] the maximum number of packets we might need to send in an averaging interval. A byte stream could result in at most one partially filled packet per period, or I_{avg} partially filled packets in the interval; all other packets must be maximum size packets. Therefore,

$$I = I_{avg} \times T \quad (4a)$$

$$n_{avg} = I_{avg} + \left\lfloor \frac{I_{avg} \times S_{avg} - I_{avg} \times 1\ byte}{s_{max}} \right\rfloor \quad (5a)$$

$$x_{ave} = \frac{I}{n_{avg}} \quad (6a)$$

To conform to these parameters, we must calculate the parameters for rate control. To allow the most data possible to be sent right away, we want $credits_0$ to be as large as possible and hence $incr_i$ must be as small as possible. Therefore, initial values are chosen as follows

$$decr_{\min} = \left\lceil \frac{S_{\min}}{s_{max}} \right\rceil \quad (7a)$$

$$incr_1 = incr_2 = \cdots = incr_{(Iavg-1)} = decr_{\min} \quad (8a)$$

$$credits_0 = n_{avg} - decr_{\min} \times (I_{avg} - 1) \quad (9a)$$

Given the assumptions made above, we can make a first approximation (not considering processing latency) to the delay parameter of the requested internetwork channel as

$$D = d_{nmax} = d_j = \frac{D_{stream} - d_{sm}}{2,} \quad (10a)$$

The loss rate for the requested network channel will be the same as the loss rate the client requested for packets

$$W = W_{err} \quad (11a)$$

(b) Traffic and performance parameters for constant-size STDUs with $STDU_{max}$ > 1

Since no packet needs to be larger than the largest possible STDU, the equation for calculating s_{max} becomes

$$s_{max} = \min(s_{net}, S_{err}, STDU_{max}) \quad (1b)$$

n_{trans} depends on the maximum number of STDUs in a period (N_{max}) as well as the minimum amount of data which should be given to the network layer in a period (S_{trans}). If there is no fragmentation, $n_{trans} = N_{max}$.[14] If there is fragmentation, we can only have one non-maximal packet per STDU (each containing at least one byte); all others must maximum size. A packet containing data from a constant-size STDU with $STDU_{max}$ > 1, may not contain data from any other STDU. Therefore, packetizing S_{trans} bytes will result in no more than

$$n'_{trans} = \left\lfloor \frac{S_{trans}}{STDU_{max}} \right\rfloor$$

non-maximal packets (essentially one per STDU). The rest of the packets must then be of maximum size. Therefore, we can calculate n_{trans} as follows,

$$n_{trans} = \begin{cases} N_{max}\ ,\ STDU_{max} = s_{max} \\ n'_{trans} + \left\lfloor \frac{S_{trans} - n'_{trans}}{s_{max}} \right\rfloor\ ,\ STDU_{max} > s_{max} \end{cases} \quad (2b)$$

Similarly, n_{avg} can be calculated as

[13] Note that n_{trans} is defined for a period, while n_{avg} is defined for an averaging interval of I_{avg} periods.

[14] Recall from section 2 that for constant-size STDUs, N_{max} is calculated from other input parameters, i.e. $N_{max} = \frac{S_{max}}{STDU_{max}}$.

$$n_{avg} = \begin{cases} N_{max} \times I_{avg} \ , \ STDU_{max} = s_{max}, \\ n'_{avg} + \left\lfloor \dfrac{I_{avg} \times S_{avg} - n'_{avg}}{s_{max}} \right\rfloor \ , \ STDU_{max} \geq s_{max}, \end{cases} \quad (5b)$$

where

$$n'_{avg} = \left\lceil \frac{I_{avg} \times S_{avg}}{STDU_{max}} \right\rceil$$

The calculation for $decr_{\min}$ also becomes more complex due to fragmentation. The minimum number of STDUs generated in a period is calculated from $S_{\min}$. This quantity is multiplied by the size of the (constant-size) STDUs to get the minimum number of bytes to be transmitted.[15] Then $decr_{\min}$ is the minimum number of packets required to transmit this number of bytes.

$$decr_{\min} = \left\lceil \frac{STDU_{max} \times \left\lfloor \dfrac{S_{\min}}{STDU_{max}} \right\rfloor}{s_{max}} \right\rceil \quad (7b)$$

Equations (3), (4), (6), and (8)-(11) are unchanged and are not repeated here.

(c) Traffic and performance parameters for variable-size STDUs with $STDU_{max} = s_{max}$

For variable-size STDUs, the equations for calculating the network-layer traffic and performance parameters are more complicated. s_{max} is calculated as in equation (1b).

If $s_{max} = STDU_{max}$ (no fragmentation), the maximum number of packets which may be sent in a period is simply N_{max},[16] so

$$n_{trans} = N_{max} \quad (2c)$$

and

$$n_{avg} = I_{avg} \times N_{max} \quad (5c)$$

Equations (3), (4), and (6) are unchanged, thus

$$x_{\min} = x_{ave} = \frac{T}{N_{max}}$$

To perform rate control, *credits* is set to N_{max} at the beginning of each period, so

$$decr_{\min} = N_{max} \quad (7c)$$

which implies that, for all i,

$$incr_i = credits_0 = N_{max}$$

Equations (8)-(11) are also unchanged and are not repeated.

(d) Traffic and performance parameters for variable-size STDUs with $STDU_{max} > s_{max}$

The number of packets which could be required to transmit the maximum allowable data in a smoothing interval of I_{sm} periods can be calculated as $n'_{sm} = I_{sm} \times N_{max}$ non-maximal packets (one per STDU) and

$$\left\lfloor \frac{I_{sm} \times S_{trans} - n'_{sm}}{s_{max}} \right\rfloor$$

maximum size packets. Then the largest number of packets which must be sent in any smoothing interval of length I_{sm} is given by

$$n'_{sm} + \left\lfloor \frac{I_{sm} \times S_{trans} - n'_{sm}}{s_{max}} \right\rfloor$$

Therefore,

$$n_{trans} = N_{max} + \left\lceil \frac{1}{I_{sm}} \times \left\lfloor \frac{I_{sm} \times S_{trans} - n'_{sm}}{s_{max}} \right\rfloor \right\rceil \quad (2d)$$

The number of packets required in an averaging interval of I_{avg} periods can be divided into $n'_{avg} = I_{avg} \times N_{max}$ non-maximal packets and

$$\left\lfloor \frac{I_{avg} \times S_{avg} - n'_{avg}}{s_{max}} \right\rfloor$$

maximum-size packets. Therefore,

$$n_{avg} = n'_{avg} + \left\lfloor \frac{I_{avg} \times S_{avg} - n'_{avg}}{s_{max}} \right\rfloor . \quad (5d)$$

Finally, $decr_{\min}$ can be calculated as

$$decr_{\min} = N_{max} + \left\lfloor \frac{S_{\min} - N_{max}}{s_{max}} \right\rfloor \quad (7d)$$

Equations (3), (4), (6), and (8)-(11) are unchanged and are not repeated here.

6.2.2.2. Allocation of resources to CM layer

The remaining CM-layer parameters are used to determine the buffer requirements of the channel.

---

[15] which may be less than $S_{\min}$, if $S_{\min}$ is not a multiple of $STDU_{max}$.

[16] Recall from section 2 that if a client requests a variable-size STDU, the number of STDUs per period is assumed to be constant.

Buffer size required at the sending host

The shared buffer B_S is provided by the user as a parameter of the channel establishment request. CM then uses the following equations to verify that the buffer provided is large enough to meet the performance requirements of the channel (i.e. to ensure that no packets are lost due to overflow of *CM* buffers). At the sender, the size of the shared buffer is determined by the requirements for holding working data, for smoothing, for allowing C_S to be early, and for compensating for alignment of STDUs in the buffer. The minimum buffer requirement, b_S, is conservatively estimated below:

$$b_S = 2 \times S_{max} \quad \textit{(current and next periods)} \quad (12)$$
$$+ S_{Sslack} \quad \textit{(workahead for } C_S\textit{)}$$
$$+ b_{sm} \quad \textit{(for smoothing)}$$
$$+ b_{align} \quad \textit{(for aligning STDUs)}$$

If the buffer provided by the client is smaller than b_S, the channel establishment fails.

- Constant-size STDUs:

For constant-size STDUs, one conservative upper bound on the amount of buffer space needed for smoothing (b'_{sm}) can be calculated by assuming that up to S_{max} bytes can be placed in the buffer during a period, while a maximum of S_{trans} bytes are removed for network transmission. Therefore, the calculation allocates $S_{max} - S_{trans}$ bytes of buffer space for each period in any smoothing interval. This analysis gives the following upper bound on b_{sm}:

$$b'_{sm} = STDU_{max} \times \left\lceil \frac{(S_{max} - S_{trans}) \times I_{sm}}{STDU_{max}} \right\rceil \quad (13b)$$

For a byte stream channel this equation simplifies to

$$b'_{sm} = (S_{max} - S_{trans}) \times I_{sm} \quad (13a)$$

Another conservative upper bound is given by the maximum amount of data which could be presented in the entire smoothing interval.

$$b''_{sm} = STDU_{max} \times \left\lceil \frac{I_{sm} \times S_{trans} - S_{trans}}{STDU_{max}} \right\rceil \quad (14b)$$

And for a byte stream

$$b''_{sm} = I_{sm} \times S_{trans} - S_{trans} \quad (14a)$$

Of these, the best upper bound will, necessarily, be the minimum of the two

$$b_{sm} = min(b'_{sm}, b''_{sm}) \quad (15)$$

Since constant-size STDUs do not have any alignment problem in the buffer,

$$b_{align} = 0 \quad (16a,b)$$

- Variable-size STDUs:

For a channel upon which variable-size STDUs are transmitted, b_{sm} is calculated in the same manner as for constant-size STDUs with $STDU_{max} > 1$. Therefore, equations (13c,d) and (14c,d) are the same as (13b) and (14b) respectively.

Additional buffer space is needed because of possible alignment problems in the buffer. We have chosen to require that an STDU be contiguous in the shared circular buffer (i.e. a single STDU cannot ‘‘wrap-around’’ from the bottom of the ‘‘circular’’ buffer to the top.) Therefore, up to $STDU_{max}$ bytes could be wasted in the buffer at any one time. Hence,

$$b_{align} = STDU_{max} \quad (14c,d)$$

Buffer size required at the receiving host

At the receiver, buffer space is needed to account for the total workahead up to that point in the stream. This includes the workahead specified across the C_S/CM_S interface (S_{Sslack}) and the workahead across the CM_R/C_R interface (S_{Rslack}). If both were positive workahead, the buffer space required would be the largest of the two. However, the workahead across the CM_R/C_R interface is in addition to the workahead specified for C_S since it defines the amount C_R can fall *behind*, not the amount CM_R can be ahead. Additional buffer space is also needed because of the delay d_j, which is introduced by CM_R to tolerate delay jitter in the network. The buffer space for compensating for delay jitter is estimated by computing the number of periods included in the time $2 \times d_j$ [17] assuming S_{trans} bytes are presented in each period. Therefore, the amount of buffer space required at the receiver can be conservatively estimated as

[17] Buffer space due to delay jitter is twice d_j because we delay the stream by d_j at the start to avoid starvation which could occur because delay jitter introduces uncertainty as to the exact starting time of the stream on the receiver, and because data could arrive as much as d_j earlier than its deadline.

$$b_R = 2 \times S_{max} \quad \textit{(current and next periods)} \quad (17)$$
$$+ S_{Sslack} \quad \textit{(workahead at the sender)}$$
$$+ S_{Rslack} \quad \textit{(amount } C_R \textit{ can be late in consuming)}$$
$$+ b_{sm}$$
$$+ b_{align}$$
$$+ S_{trans} \times 2 \times \left\lceil \frac{d_j}{T} \right\rceil \quad \textit{(compensate for delay jitter)}$$

where b_{sm} and b_{align} are as calculated for b_S. Equation (17) gives an upper bound on the amount of buffer space needed to protect against packet loss in the CM buffers. However, there is no reason for CM_R to know S_{Rslack}, which is chosen by C_R. Therefore, CM_R will calculate b'_R, the amount of buffer space required at the receiver without considering workahead:

$$b'_R = 2 \times S_{max} \quad \textit{(current and next periods)} \quad (18)$$
$$+ S_{Sslack} \quad \textit{(workahead at the sender)}$$
$$+ b_{sm}$$
$$+ b_{align}$$
$$+ S_{trans} \times 2 \times \left\lceil \frac{d_j}{T} \right\rceil \quad \textit{(compensate for delay jitter)}$$

This value is passed on to C_R, who then calculates

$$b_R = b'_R + S_{Rslack} \quad (19)$$

If C_R is unable to allocate a buffer of this size or greater, the channel establishment fails.

The processing requirements of the CM entities can be calculated as a function of a fixed amount of processing each time CM runs, CPU_f, and a variable amount, CPU_v, which depends on the number of packets sent each time CM runs. The maximum delay between the beginning of a period and the time CM runs in that period, d_{CM_S}, must be estimated and included in D_{stream}. Thus, equation (10) must be extended to become

$$D = d_{nmax} = \frac{D_{stream} - d_{sm} - d_{CM_S} - CPU_{S_{max}}}{2} \quad (10')$$

where $CPU_{S_{max}} = CPU_f + CPU_v(n_{trans})$ at the sender.

6.2.2.3. Acquisition of network channel

CM requests a network layer channel with the parameters given by equations (1), (3), (4), (6), (10'), and (11) above and passes $STDU_{max}$, *CONST_LEN*, T, S_{max}, S_{trans}, I_{sm}, d_j, S_{avg}, I_{avg}, S_{err}, *REPLACE*, and *Dummy* on to CM_R. At the receiver, CM_R then makes its CPU reservation and checks that

$$(d_{CM_R} + CPU_{R_{max}}) \leq D_{remaining}$$

remaining delay, where d_{CM_R} is an estimate of the maximum CPU latency on the receiving host, and CPU_{max_R} is an estimate of the maximum processing time for CM_R. As stated previously, CM_R calculates a suggested buffer size, b'_R which it passes on to C_R. If C_R is able to allocate the required buffer (of size $b_R = b'_R + S_{Rslack}$), and sufficient CPU resources are available on the receiving host, the establishment succeeds.

6.2.3. Actions during data transfer

This section describes the actions of each of the entities while data transfer is occurring.

- C_S:

1. Indicate new stream via `cmOpenStream()` call
2. Check for room in buffer, either by checking descriptors or by checking if

$$(current_p - N_{read} + Buffer.size) \bmod Buffer.size < data\_size.$$

3. Put data in shared buffer
4. Update flags in descriptor and $N_{written}$ to indicate new data is present
5. Goto 2 until end of stream
6. Indicate end of stream via `cmCloseStream()` call
7. On initiation of a new stream, goto 1

- CM_S:

Upon `cmOpenStream()` call from C_S:

1. Read clock and record time of `cmOpenStream()` call (t_{start})
2. Verify and record stream parameters
3. Determine time at the beginning of the first period ($t_0 = t_{start} + d_{startup}$)
4. Send OPEN_PDU to CM_R including $d_{startup}$
5. Set $credits = credits_0$; initialize $incr_0$, $incr_2$, ..., $incr_{(Iavg-1)}$
6. Schedule self to run again at beginning of the first period

$$nextTime = t_0 = t_{start} + d_{startup}$$

After startup delay:

7. Compute the number of STDUs in the buffer (*NumSTDUs*).[18] Packetize all complete STDUs in the buffer as specified by the following pseudo-code:

```
[*  This pseudo-code will determine
 *  packet boundaries for all STDUs
 *  in the buffer and schedule the
 *  process to run again the next
 *  time it can legally send more data.
 *]

Initialize(Smin:integer; .... )
begin
    decr_min := f(Smin);
    credits := credits_0;
    for i := 0 to (Iavg - 1) do
   incr[i] := decr_min;
    ........................
end;

[* The function Packetize takes data
 * from the shared buffer determines
 * boundaries for network packets.
 *
 * numSTDUs is the number of STDUs in
 *    the buffer
 * BufPtr points to the next STDU in
 *    the buffer
 * decr_min is the same as in
 *    Figure 6.4
 * current is the number of the next
 *     STDU to be packetized
 *]

Packetize(numSTDUs:integer;
   BufPtr:^STDU; credits; decr_min;
   current:integer)

numPackets: integer;
     [* num pkts in period *]
i:integer;

begin
  [* packetize all STDUs in buffer *]
  while (NumSTDUs > 0) do
  begin
    [* STDUs for "current" period *]
    numPackets := 0;  [* reset cnt *]
      while ((credits > numPackets)
                and (NumSTDUs > 0)) do
      begin
        Make_next_packet(BufPtr);
        Update_BufPtr();
        numPackets := numPackets + 1;
        if (last_packet_of_STDU) then
          NumSTDUs := NumSTDUs - 1;
      end;

      [* set up state to either run in
       * next period (if all STDUs in
       * buffer have been packetized)
       * or continue  packetizing using
       * credits from the next period
       *]

       [* time in next period *]
       nextTime := nextTime + T;

      [* add credits from next period *]
      decr := max(decr_min, numPackets);
      credits = credits - decr
         + incr[current];
      incr[current+Iavg-1] := decr;
      current := current + 1;
    end;
  end;
```

8. Update descriptor flags and N_{read}
9. Schedule self to run again at *nextTime*, continuing at step 7.

Upon receipt of a `cmCloseStream()` call:

10. Record current ending location of data in the buffer.
11. Continue steps 7-9 until data is sent up to location recorded in 10.
12. Send CLOSE_PDU to CM_R
13. Sleep until next `cmOpenStream()` call.

- CM_R:

Upon receipt of an OPEN_PDU from CM_S:

1. Read clock to establish beginning of stream (τ_{start})
2. Record new stream parameters
3. Indicate beginning of stream to C_R by returning from `cmWaitForOpen()` call, passing

[18] For a connection which is used to transport variable-size STDUs, the buffer descriptors must be checked to determine the number of STDUs in the buffer. In that case, the next descriptor is checked for a complete STDU instead of checking whether *NumSTDUs* > 0. For constant-size STDUs, the number of STDUs in the buffer can be inferred from $N_{written} - N_{read}$.

$delay = d_{startup} + d_j$ as a return parameter.

4. Record starting time of stream on receiving host:

$$\tau_0 = \tau_{start} + delay$$

5. Sleep until DATA_PDU or CLOSE_PDU arrives.

Upon receipt of DATA_PDU:

6. Check for room in shared buffer by checking whether

$$(current_p - N_{read} + data\_size) \bmod Buffer.size < Buffer.size.$$

If there is room, put data into shared buffer; otherwise throw away the new data. If data is corrupted and *REPLACE* has been defined to be true, replace data with the *Dummy* data defined at channel establishment.

7. Update flags indicating condition of new data (and possible packet losses discovered upon receipt of new data) and $N_{written}$.

Upon receipt of CLOSE_PDU:

8. Record ending location of data in the buffer.
9. Indicate end of stream to C_R by returning from `cmWaitForClose()` call, passing data location recorded above as a return parameter.
10. Sleep until next OPEN_PDU

- C_R:

Upon returning from `cmWaitForOpen()`:

1. Make non-blocking call to `cmWaitForClose()`.
2. Begin reading data from buffer after *delay* has elapsed
3. Change flag bits and N_{read} when data has been read

Upon return from `cmWaitForClose()`:

3. Read data up to location specified in `cmWaitForClose()` return (as specified in step 8 of CM_R actions).
4. Make blocking call to `cmWaitForOpen()`.

6.3. Examples of local adaption layers for unsophisticated clients

The service is designed to be used directly by CM applications, however some applications may not be able to or desire to perform all the functions required to access the service. This section will briefly demonstrate the universality of the interface we have described, by presenting two sample interfaces for such clients, which may be implemented on top of CMTS with little difficulty.

6.3.1. DMA client

A client may wish to transfer data into the shared buffer via DMA without requiring a CPU interrupt per period or per STDU. A thin layer is needed to manage the structured buffer. The channel would be set as a byte stream and the buffer would be split in two, with the DMA device writing into one half at a time. When it fills its half, the CPU is interrupted, causing the local adaption layer to run again. The actions of this layer are described below:

At beginning of stream:

1. Configure DMA device to write into top half of buffer. Check for room in buffer.

2a. Indicate new stream via `cmOpenStream()` call to CM_S

3. Wait for interrupt from DMA device.

Upon interrupt from DMA device:

4. Set flags of filled buffer to indicate data is present and update $N_{written}$
5. Check that other half of buffer is empty

6a. If not, block DMA device and sleep for one period and check again until other half of buffer is empty.

6b. When second half is empty, configure DMA device to write into other half

7. Sleep until next interrupt from DMA device (continue at step 4) or `close()` call from the user to close the stream (continue at step 8).

Upon close request from user:

8. Indicate end of stream to CM_S via `cmCloseStream()` call.
9. On initiation of a new stream, goto 1

The steps taken by a local adaption layer at the receiver for a similar device would be analogous and are not given here.

6.3.2. Send/Receive interface

Those clients which are user processes generating data for transmission may prefer a more familiar send/receive interface with variable-size messages. This service will use a channel of variable-size STDUs. Copying is used to emulate standard send/receive buffer management semantics. The advantage for CM clients of using this scheme over a message scheme is that smoothing and rate control are performed automatically by the service, and user-kernel interactions are diminished (*send*() causes a user-level library routine to be invoked, but may not result in a kernel call. The kernel call is unnecessary because CM_S will check the shared buffer for data once each period.)

Upon *send* call by the user:

1. Check whether there is room in the shared buffer for this new STDU (as described above).

2a. If not, sleep for some time (e.g. T / N_{max}) and check again, until enough buffer space is freed.

2b. Copy from the user-supplied buffer into the shared buffer.

3. Update flags in buffer and $N_{written}$

4. Return to user

Upon a *receive* call at the receiver:

1. If no data is present, the receive returns with zero data.

2. Copy from shared buffer into the user-supplied buffer.

7. Summary

We have presented a design and implementation considerations for a Continuous Media Transport Service (CMTS), a data transport service designed specifically for delivery of continuous media. This service takes advantage of the higher predictability of CM streams to (potentially) provide the same service while allocating less network resources. In addition, CMTS provides a logical stream abstraction to aid in managing data transmission on a CM channel and an error-handling mechanism, which can be adapted flexibly to the typical demands of CM applications.

The basic concepts introduced, such as the notion of stream data unit, as well as the large variety of parameters offered at the service interface can be used by communicating CM applications for a relatively flexible characterization of (e.g. voice or video) streams. This flexibility is also provided for the mapping of STDUs onto packets of an underlying network service (1:1, fragmentation or concatenation as options), where the mapping may even be controlled by the transport service users (e.g. to limit the negative consequences of a data loss).

We should note that the solution chosen also does offer the possibility to allow a (de-) coding process to react to the state of the communication system, as suggested e.g. in [GiG91]. The communication system's state might be considered to be reflected by the actual occupancy of the shared buffers (B_S and B_R) on the sending and receiving end-systems, which could lead to a variation of the (de-)coding rate.

For a completion of the present design it will still be necessary to integrate the experiences gained in the prototype implementation of CMTS in an extended service/protocol design. The extensions will have to specify, in particular, additional possibilities of reacting to protocol errors as well as the exchange of different types of control information.

Limitations of the solution primarily concern the buffer requirements in the end-systems, which may become significant in those cases, when delay jitter within the network and within the end-systems will become large and additionally large traffic fluctuations exist within the arrival process of the stream. However we believe that in future computer systems (even in workstations and personal computers) we can expect provision of communication buffers in the range of (a few) MByte at least for CM applications, if this yields to significant simplifications and performance improvements. Realization of multi-point connections (e.g. required in video-conferencing) by means of (a possibly large number of) point-to-point connections, which would be the solution based on the CMTS service, may also lead to some inefficiencies.

In parallel to the CMTS prototype implementation, presently a modeling study is carried out in order to get some insight in the impact, which configuration parameters of end-systems (such as buffer sizes, run-times of communication software, etc), properties of the underlying network service (such as packet delay jitter, packet loss rate, etc.) and the local load of the end-

systems may have on the quality of the CMTS service as observed by CM clients (e.g. expressed by the probability of a buffer overflow with resulting loss of data and/or by the probability of late arrival of data in the B_R buffer).

8. Acknowledgements

The authors would like to express their particular gratitude to Amit Gupta and Francesco Maiorana for their engagement in the implementation of the CMTS prototype and to Eckhardt Holz for his detailed simulation study to analyze the behavior of the CMTS service under various boundary conditions. In addition, Amit Gupta contributed to the design of the credit scheme for rate control, and the correction and revision of several formulae.

A large number of in-depth discussions with a lot of resulting stimuli have taken place during the CMTS design within Tenet research team at International Computer Science Institute and University of California/Berkeley. In particular, Prof. Domenico Ferrari as head of Tenet team and the group members Riccardo Gusella, Bruce Mah, Hui Zhang and Dinesh Verma have provided very valuable suggestions during the preparation of this report. This support is sincerely acknowledged by the authors.

Special thanks also go to Prof. David Anderson and Ramesh Govindan for their comments which helped to improve an earlier version of this report.

9. References

[And90] D. P. Anderson, ‘‘Meta-Scheduling for Distributed Continuous Media’’, UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, (Oct. 1990).

[AGH90] D. P. Anderson, R. Govindan, G. Homsy, ‘‘Design and Implementation of a Continuous Media I/O Server’’, Proc. 1st Internat. Workshop on Network and O.S. Support for Dig. Audio and Video, ICSI TR-9-062, International Computer Science Inst., Berkeley, CA, (Nov. 1990).

[AHS90] D. P. Anderson, R. Herrtwich, C. Schaefer, ‘‘SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet,’’ Int. Comp. Sci. Inst., Technical Report No. ICSI TR-90-006 (1990).

[BaM91a] A. Banerjea, B. Mah, ‘‘The Real-Time Channel Administration Protocol’’, Proc. 2nd Int. Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg (November, 1991).

[BaM91b] A. Banerjea, B. Mah, ‘‘The Design of a Real-Time Channel Administration Protocol,’’ internal document (1991).

[Che88] G. Chesson, ‘‘XTP/PE Overview,’’ 13th Conf. on Local Computer Networks, IEEE Comp. Soc. (October 1988), 292-296.

[ChW89] D. R. Cheriton, C.L. Williamson, ‘‘VMTP as the Transport Layer for High-Performance Distributed Systems,’’ IEEE Commun. Magazine, Vol. 27, No. 6 (1989), 37-44.

[CLZ87] D. D. Clark, M. L. Lambert, L. Zhang, ‘‘NETBLT: A High-Throughput Transport Protocol,’’ ACM SIGCOMM Workshop on Frontiers in Computer Netw. (1987).

[Cru87] R. L. Cruz, ‘‘A calculus for Network Delay and a Note on Topologies of Interconnection Networks’’, Ph.D. Dissertation, Report no. UILU-ENG-87-2246, University of Illinois, (July 1987).

[DDK90] W.A.Doeringer, D. Dykeman, M. Kaiserswerth, B.W. Meister, H. Rudin, R. Williamson, ‘‘A Survey of Light-Weight Transport Protocols for High-Speed Networks,’’ IEEE Trans. on Commun., Vol. 38, No. 11 (1990), 2025-2039.

[FeV90] D. Ferrari and D. Verma, ‘‘A Scheme for Real-Time Channel Establishment in Wide-Area Networks,’’ IEEE J. Sel. Areas in Comm. SAC-8 (April 1990).

[GiG91] M. Gilge and R. Gusella, "Motion Video Coding for Packet Switching Networks: An Integrated Approach," SPIE Conf. on Visual Commun. and Image Processing, Boston (November, 1991).

[GoA91] R. Govindan and D. Anderson, ‘‘Scheduling and IPC Mechanisms for Continuous Media,‘‘ Proc. of Sym. on Operating System Principles, pp.68-80, October 1991.

[HTH89] P. Haskell, K. H. Tzou and T. R. Hsing, "A Lapped-Orthogonal-Transform Based Variable Bit-Rate Video Coder for Packet Networks," Int. Conf. on Acoustics, Speech and Signal Proc., Glasgow, Scotland, May 23-26, 1989.

[HSS90] D. Hehmann, M. Salmony, H.J. Stuettgen, ‘‘Transport Services for Multi-Media Applications on Broadband Networks,’’ Computer Commun., Vol. 13, No. 4 (1990), 197-203.

[ITC91] Proc. Workshop on ‘‘Continuous Time Media’’, Information Technology Center, Carnegie Mellon University, Pittsburgh (June, 1991).

[LaS91] T. F. La Porta, M. Schwartz, ‘‘Architectures, Features, and Implementation of High-Speed Transport Protocols,’’ IEEE Network Magazine, Vol. 5, No. 3 (1991), 14-22.

[Leg91] D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," Commun. of the ACM, Vol. 34, No. 4, (1991).

[LiH91] M. Liebhold, E. M. Hoffert, ‘‘Toward an Open Environment for Digital Video,’’ Commun. ACM, Vol. 34, No. 4 (1991), 104-112.

[NRS90] A.N. Netravali, W.D. Roome, K. Sabnani, ‘‘Design and Implementation of a High Speed Transport Protocol,’’ IEEE Trans. on Commun., Vol. 38, No.11 (1990), 2010-2024.

[Wat89] R. W. Watson, ‘‘The Delta-t Transport Protocol: Features and Experience,’’ Proc. IFIP workshop on Protocols for High-Speed Networks, North-Holland (1989), 3-18.

[Wol81] B. Wolfinger, ‘‘Das Modellierungssystem MOSAIC zur Analyse und Optimierung hierarchisch organisierter Kommunikationsprotokolle in Rechnernetzen,’’ Elektronische Rechenanlagen, Vol. 23, No. 5 (1981), 199-211 (in German).

[WrT90] D. J. Wright, M. To, ‘‘Telecommunication Applications of the 1990s and their Transport Requirements,’’ IEEE Network Magazine, Vol. 4, No. 2 (1990), 34-40.

[ZhV91] H. Zhang and D. Verma, ‘‘Design Documents for RTIP/RMTP,‘‘, internal document (1991).

[Zit91] M. Zitterbart, ‘‘High-Speed Transport Components,’’ IEEE Network Magazine, Vol. 5, No. 1 (1991), 54-63.