Submitted for publication

Loss Profiles : A Quality of Service Measure in Mobile Computing[1]

Krishanu Seal and Suresh Singh

Corresponding Author

Suresh Singh
Department of Computer Science
University of South Carolina
Columbia, SC 29208
email: singh@cs.scarolina.edu
Tel: 803-777-2596
Fax: 803-777-3767

With rapid technological advances being made in the area of wireless communications it is expected that, in the near future, *mobile users* will be able to access a wide variety of services that will be made available over future high-speed networks. The quality of these services in the high-speed network domain can be specified in terms of several QOS parameters.

In this paper we identify a new QOS parameter for the mobile environment, called *loss profiles*, that ensures *graceful degradation* of service (for applications that can tolerate loss) in situations where user demands exceed the network's capacity to satisfy them. A new transport sub-layer is proposed that efficiently implements this new QOS parameter. We also show how this protocol can be easily incorporated into existing proposals for high-speed network transport layer protocols.

1 Introduction

Wireless computing (or *mobile* computing) refers to an emerging new computing environment incorporating both wireless and high-speed networking technologies. In the near future, it is expected that millions of users equipped with *personal digital assistants* (palm-top computers with wireless communications technology) will have access to a wide variety of services that will be made available over national and international communication networks. Mobile users will be able to access their data and other services such as electronic

[1]This work was supported by the NSF under grant number NCR-9410357

mail, electronic news including special services such as stock market news, videotelephony, yellow pages, map services, electronic banking, etc., while on the move.

What are some of the issues related to providing the different types of services discussed above? Two broad issues that need to be considered are the following:

- Design of an efficient network *architecture* to support mobility and protocols to track users in order to provide uninterrupted service.

- Developing *transport protocols* for the mobile environment with an eye towards maintaining quality of service guarantees for open connections as users move about.

A typical network architecture supporting wireless computing is illustrated in Figure 1. Here a geographical region such as a campus is divided into *microcells* with a diameter of the order of hundreds of meters. All mobile users within a microcell communicate with a central host machine within that cell who serves as a gateway to the wired networks; this machine is called a *mobile support station* (MSS). The MSSs are also responsible for tracking mobile users and forwarding data to the new location of the mobile users. In addition to these functions, the MSSs also provide commonly used applications software to mobile users who can download it and run it locally or run it remotely on the MSS itself (see Badrinath[6]). MSS nodes are connected to one another and to commercial high-speed networks thus providing *universal* connectivity to all mobile users, Figure 2. Thus mobile users will be able to set up connections with other mobile or fixed hosts and continue to receive service along these connections as they move about. Some of the protocols to *maintain* connections in the face of mobility are discussed in Singh[4] and Teraoka[16].

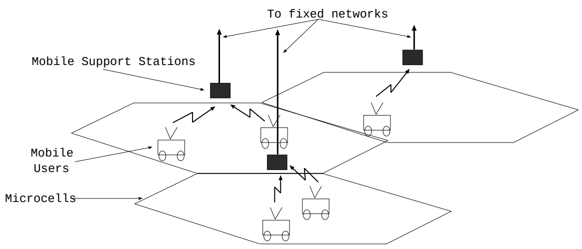


Figure 1: Microcellular architecture.

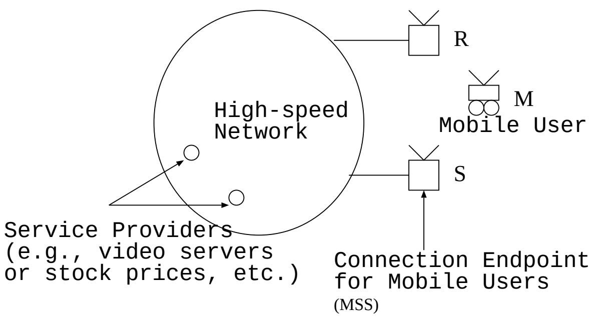


Figure 2: Connectivity to fixed high-speed networks.

In order to support the wide range of services discussed earlier, future (i.e., 3*rd* generation) wireless networks will have to be considered extensions of fixed (or wired) networks. This implies a need for developing new networking protocols suited to the wireless environment. In this paper we focus on developing transport protocols for wireless networks. Why do we need to develop different transport layer protocols for the wireless environment? There are two reasons why this needs to be done:

- Traditional protocols such as TCP are very inefficient if used unchanged in the mobile environment. This is because in TCP the sender begins retransmission of packets if they are not acknowledged within a short amount of time (hundreds of milliseconds). In a mobile environment, as a user moves between cells, there is a brief blackout period while the mobile unit performs a 'handshake' with the new MSS. This period can be as long as 1 second thus delaying the transmission of acknowledgements for packets received. This results in the TCP sender timing out and retransmitting the unacknowledged packets thus greatly reducing the efficiency of the connection. A solution to this problem is the I-TCP protocol, a modified TCP protocol, implemented at Rutgers University as part of the DataMan project (see Barke[1]), that provides efficient reliable communication for the wireless environment.

  A benefit of the above implementation is that it allows mobile hosts to be connected over the Internet. However, in a real world implementation of a mobile network, it is probably a better idea to use *virtual circuits* at the network layer. This avoids the problem of end-to-end retransmissions and can more effectively deal with mobility issues. One such solution has been proposed in Singh[4].

- Future applications to be provided to mobile users will include audio (e.g., telephone, audio conferencing, etc.) and video applications (e.g., map information, viewing movies, etc.). These applications have real-time constraints and therefore cannot be implemented using the I-TCP or UDP protocol. Transport protocols proposed for high-speed networks (such as the Multistream Protocol or MSP, see Schwartz[8]), that provide numerous features to support such applications, cannot be adapted to the mobile environment because *the bandwidth availability can vary over the lifetime of a connection* in mobile environments. This is caused because users in a mobile environment are mobile and unpredictable in their movement.

  To illustrate a consequence of this unpredictable mobility, consider a situation where several mobile users have opened high-bandwidth connections. When these connections are set up, the network ensures that the users receive some guaranteed bandwidth resulting in some well defined quality of service (such as max. loss probability, bounded delay, etc.). Since these users are all mobile, it is possible that many of them could move into the same cell. In such a situation, it is very likely that the available bandwidth of the cell will be exceeded resulting in the original QOS (quality of service) parameters being violated. This situation does not arise in high-speed networks because users are not mobile during the life-time of a connection.

The focus of the research presented here is to develop a new transport layer protocol and associated quality of service parameters to effectively deal with the problem caused by the unpredictable mobility of users. In the next section we discuss the new quality of service measure, *loss profiles*, in a general way and present experiments that illustrate the need for such a QOS parameter. Section 3.2 presents our specification for a new transport sub-layer, called LPTSL (Loss Profile Transport Sub-Layer), that can efficiently implement loss profiles. Section 5 shows how loss profiles can be easily incorporated into the MPEG-2 specification and section 4 shows how this can be done for the Multi-stream Protocol of Schwartz[8]. We consider MPEG-2 specifically because we expect many future video-based applications to be based on the MPEG-2 standard. Section 6 discusses our current work and future directions.

2 Need for Loss Profiles

To better explain the need for this new QOS parameter (i.e., loss profiles) let us consider an example. Let us assume that the total bandwidth available in a cell is 32kbps and that initially the cell has two mobile users, M1 and M2, each with an open constant bit-rate connection operating at 16kbps. Now a third user

M3, also with a open constant bit-rate connection of 16kbps, enters the cell. Since the requested bandwidth (i.e., 48kbps) exceeds the available bandwidth, either one of the connections needs to be terminated or all connections must see degraded quality, i.e., receive a smaller bandwidth. We believe that the latter approach is the fair one because it is not easy to prioritize connections that different users may have.

Let us assume that all three connections in our example now receive a data rate of 10.66kbps. If the connections opened by the three users were all data connections (i.e., delays are not very important but all data needs to be delivered correctly) then the effect of the slower data rate is simply longer delays. If, on the other hand, one of the users had a video connection (e.g., television) the smaller data rate could cause serious problems. The video connection was set up as a 16kbps connection and therefore data will arrive at the MSS at the rate of 16kbps. Now, since the user only has 10.66kbps available, the MSS will discard 5.34kbits of data per second. If this data is discarded indiscriminately, the user may see no video at all. This is because video is typically transmitted in compressed form (either using MPEG or JPEG compression technology) and if some portion of the compressed data stream is discarded, the decompression algorithm will lose synchronization and fail to produce any video whatsoever.

One way to remedy this situation is for the mobile user or the MSS to inform the service provider (at the other end of the connection) of the reduced bandwidth. That is, request the service provider to use a higher compression ratio to deliver the video service. This is not an attractive approach because the situation where the requested bandwidth exceeds the available bandwidth is probably a **temporary** one that will be alleviated as soon as the mobile user moves into a different cell. A more serious problem with this approach is the need for the service provider to be able to increase or decrease the compression ratio by arbitrary amounts on demand *and for brief time periods* (as small as a few seconds – the cell latency of a mobile).

Our approach is based on the philosophy that once service parameters have been established for a connection they should not be renegotiated. Rather data, for the connection that is to be penalized, must be discarded **intelligently** by the MSS[2]. In other words:

- The *structure* of the data stream must be considered when data is discarded by the MSS. Suppose, as above, that the connection being penalized is a video-based connection and that the compression being used is JPEG (i.e., each frame is compressed individually). Figure 3 illustrates what the data stream for this connection may look like (the *runlength* contains positional information of the *coefficients*). If we discard data from the runlength part of the data stream, we cannot reconstruct that frame. On the other hand discarding data from the coefficient part only causes the reconstructed image to be blurred. Therefore, our approach would call for data to be discarded from the coefficient part of the data stream only. In the case of MPEG video, only the bidirectional (i.e., **B**) frames or the predicted (i.e., **P**) frames must be discarded and not the index (i.e., the **I**) frames (see Hung[5]).

- *Viewer perception* must be taken into account when data is discarded for a connection. In other words, there are different ways of discarding $x\%$ of data. The 'best' way will depend upon the nature of the application. For example, a user viewing a movie may prefer losses to manifest themselves as random frame loss (resulting in short freezeups) as opposed to 'clustered' loss every 1 second (i.e., several consecutive frames are lost). It is noteworthy that depending upon the specific application, users may prefer different types of loss. In section 2.1 we discuss this further.

Based on our discussion we have the following definition for the loss profile QOS parameter.

> **Definition 1** A **loss profile** is a description, provided by the application, of an 'acceptable' manner in which data for its connection may be discarded in the event of a bandwidth reduction at the wireless end of its connection.

[2] **It is important to note that loss profiles can only be used with applications that are tolerant of loss.**

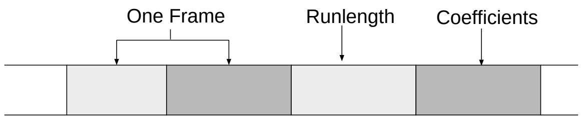


Figure 3: JPEG data stream.

2.1 Study of Viewer Perception

Data loss, implemented via loss profiles, manifests itself in a variety of ways to the end user. Thus, it is likely that some loss profiles may be more acceptable than others for specific applications. In this section we discuss results of a viewer perception study to illustrate this idea.

A set of experiments was conducted to study user responses on a video clipping subjected to varying degrees of losses. A 40 second video clip captured at approximates 16 frames per second was edited in six different ways to obtain different kinds of losses. We obtained two sets of clippings - one with 20% of the image frames discarded and the other with 40% being discarded. Each set had three different types of loss distributions, namely,

1. Clustered loss around every 5th frame – discard every 5th frame and copy the 4th frame to the 5th for 20% loss. For 40% loss, discard the 4th and 5th frames and copy the 3rd frame to these positions. Observe that to a viewer, the video will appear to freeze. The freeze interval for 20% loss is about 66 msec and is 132 msec for 40% loss.

2. Clustered loss around every 15th frame – for 20% loss, discard frames 13 to 15 and replace them with the 12th frame. For 40% loss, discard frames 10 to 15 and replace them with the 9th frame. The freeze up intervals here are about 200 msec for 20% loss and 400 msec for 40% loss.

3. Random loss of frames every 15 frames – discard 3 frames chosen randomly every 15 frames for 20% loss and 6 frames, chosen randomly, for 40% loss. A discarded frame is replaced by its predecessor (undiscarded) frame.

A group of 26 people were shown 13 different pairs of video clips. The results of the experiments are summarized in Table 1. The most interesting result of the experiment is that viewers preferred a 40% random loss to a 20% clustered loss every 15th frame (see row 13 in Table 1). This is probably because a 20% clustered loss corresponds to a freeze up period of 200 msec that is annoying to the human eye. The implication of this result is that, in a bandwidth crunch, we can get away with discarding up to 40% of user data (as opposed to only 20%). Another interesting observation is that viewers considered a random loss to be as good as a clustered loss every 5th frame (see rows 5 and 11). This result is encouraging from an implementation standpoint because, in practice, it is simpler to implement clustered losses as opposed to random losses.

Although the results indicate that certain loss distributions are clearly more desirable that others, there are some cases where users might exercise their own preference in selecting what kind of losses are acceptable to his or her taste. At any rate, we can conclude that there is a need for a mechanism that enables a user to specify a loss profile for a particular application.

2.2 Summary

We have illustrated the need for users to be able to specify loss profiles. Furthermore, we note that such a specification needs to contain at least two components:

1. Certain parts of the data stream cannot be discarded (e.g., runlength information in JPEG data) and there needs to be a mechanism to indicate this, and

Comparison#	a	b	a>b	b>a	a=b	Overall winner
1	original	20% every 5th	16	6	4	a>b
2	original	20% every 15th	20	4	2	a>b
3	original	20% random	22	3	1	a>b
4	20% every 5th	20% every 15th	20	3	3	a>b
5	20% every 5th	20% random	9	9	8	-
6	20% every 15th	20% random	5	17	14	b>a
7	original	40% every 5th	22	3	1	a>b
8	original	40% every 15th	24	1	1	a>b
9	original	40% random	20	0	6	a>b
10	40% every 5th	40% every 15th	23	3	0	a>b
11	40% every 5th	40% random	5	8	13	a=b
12	40% every 15th	40% random	4	17	5	b>a
13	20% every 15th	40% random	8	11	7	b>a

Table 1: Viewer perception study results.

2. The distribution of losses makes a big difference and needs to be specified as well.

3 Location of Loss Profiles in the Network Architecture

In this section we attempt to answer two related questions:

- where, in the network protocol hierarchy, can loss profiles be implemented most effectively?
- what are some architectural considerations that make this task easier?

In the next section we discuss the inadequacy of traditional mobile network architectures and propose a modified architecture that works better. Later, we address the problem of incorporating loss profiles within the network protocol hierarchy.

3.1 Network Architecture Considerations

Traditional mobile network architectures (see for example Badrinath[6] and Maguire[7]) are illustrated in Figure 1. The MSS nodes provide network connectivity to mobile users and are responsible for tracking users as they move and forwarding messages. We believe that this architecture is inappropriate for implementing loss profiles. This is because,

1. Consider a situation where a mobile user moves between several crowded cells (thus requiring data to be discarded in each cell). In such a situation it is likely that, *the cumulative loss* over a long time period will not match the specified loss profile (in terms of the *distribution* of losses). This is because individual MSS nodes are unaware of the past history of a connection (i.e., before the user moved into its cell) and therefore bases its decision to discard data on local information only (i.e., bandwidth limitations, other user needs, etc.). Note that requiring the service-provider (from the fixed high-speed network) to implement loss profiles for mobile users is not an acceptable solution for reasons discussed earlier in this section.

2. Loss profiles need to be implemented at the transport layer and, in addition, require the node to buffer large amounts of data (so that data can be discarded so as to follow the loss profile). This adds significantly to the complexity of MSS nodes.

3.1.1 Summary of our proposed Architecture

Our proposed architecture is discussed in detail in Singh[14]. We summarize the main points in this section. The architecture of our system may be viewed as a three-level hierarchy. At the lowest level are the mobile users (MH) who communicate with MSS nodes in each cell. Several MSSs are controlled by a machine called the Supervisor Host (SH). The SH is connected to the wired network and it handles most of the routing and other protocol details for the mobile users. In addition it maintains connections for mobile users, handles flow-control and is responsible for maintaining the negotiated quality of service. A single SH may thus control all MSS nodes within a small building.

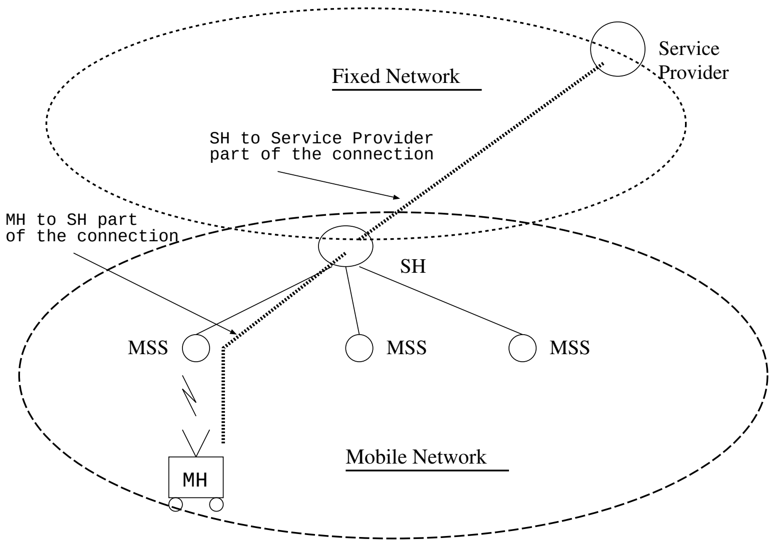


Figure 4: Connection between a mobile user and a service provider.

Our architecture separates the mobile network from the high-speed wired network and provides connectivity between the two via supervisor hosts who serve the function of a **gateway**. Thus, a mobile user may set up connections where the other end-point is either another mobile user or a fixed host (e.g., a service-provider) in the fixed network. In either case the connection is managed by the current SH of the mobile host(s). Figure 4 illustrates one such connection. As we can see the connection between the MH and a fixed host is broken in two – one between the MH and a SH and another between the SH and the fixed host. The reason for splitting the connection between the MH and the service-provider is to shield fixed nodes from the idiosyncrasies of the mobile environment.

This architecture is well suited to implementing loss profiles because a mobile user typically remains within the domain of a single SH for a large amount of time. Thus, the SH can ensure that, as the mobile user moves about between crowded cells, its cumulative losses follow the specified loss profile. In addition, since a SH manages many MSSs, it is reasonable to centralize the complexity within the SH and make the MSS nodes simple devices. We discuss flow-control and connection management issues within our architecture in Singh[4]. In the remainder of this paper we assume our architecture, though our results are applicable, with few or no changes, to the traditional architectures as well.

3.2 LPTSL: Loss Profile Transport Sub-Layer

Implementing loss profiles to behave as described presents us with a design problem. While on the one hand it is clear that, in order to discard data intelligently, the SH needs to know about the structure of the data stream. On the other hand, it is both, unreasonable and undesirable, to expect the SH to know about the encoding strategies used by different applications. Therefore the best compromise, we believe, is for the data stream to be viewed by the SH as being composed of *logical segments*. Each logical segment may be discarded completely or not be discarded at all. The application generating the data stream defines segment boundaries.

The SH can now view the data arriving for each connection as shown in Figure 5. Note that the data stream is divided into logical segments by inserting *flags* into the stream. Thus, when data needs to be discarded, the SH simply discards a segment. If the application generates MPEG data, each segment may be either a **I, B** or **P** frame. If the data is audio data, each frame may represent 20ms of audio during a talk spurt.

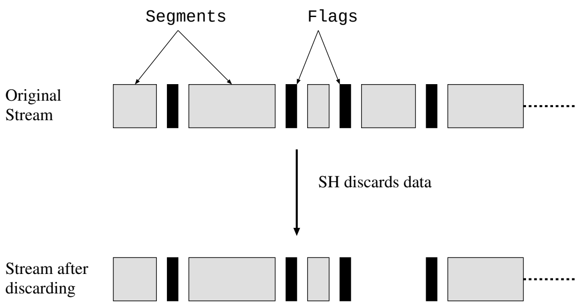


Figure 5: Data stream at SH composed of *logical segments*.

Where in the network hierarchy should loss profiles be implemented? In the realm of networked applications, the applications user worries little about the network intricacies about how to send data to a remote destination. These details are handled by lower layers of the network protocol. The transport layer in particular plays a significant role in this process. It takes over the communication details and provides certain "services" to the user. The user simply requests the transport layer for a connection endpoint with certain QOS characteristics and writes data to this connection. The transport layer delivers the data to the destination without 'interpreting' the semantic content of the data.

Loss profiles are application dependent and are necessarily defined w.r.t. the contents and structure of the data stream. Thus, loss profiles cannot be implemented at the transport layer. We propose that loss profiles be implemented immediately before the transport layer as shown in Figure 6. This layer is called the **Loss Profile Transport Sub-Layer** (or, **LPTSL**) and it includes all the functionality required for implementing loss profiles.

As discussed in section 3.1.1 we view the mobile network as consisting of a three layer hierarchy with the supervisor hosts (SHs) serving as a gateway to the fixed high-speed networks. Thus, every connection between a MH and a application provider in the fixed networks may be viewed as shown in Figure 7. The SH serves as the discarding station in the event of a bandwidth crunch, as shown in Figure 7.

Figure 8 illustrates the functionality of LPTSL in our model. A data stream is generated by the application in the fixed network. This stream is divided into logical segments by the LPTSL. The LPTSL at the supervisor host discards segments from this stream, if necessary, and passes the truncated stream to the transport layer for transmission to the mobile host. The flags are not discarded because they are frequently necessary for synchronization (as in MPEG-2, discussed in section 5.1.4) or for the purpose of informing the MH of the location of the losses.

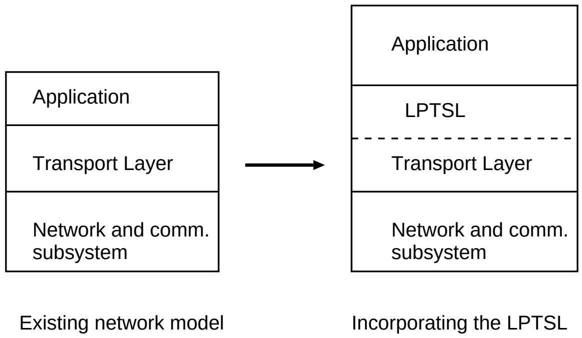


Figure 6: Position of LPTSL in the network protocol hierarchy.

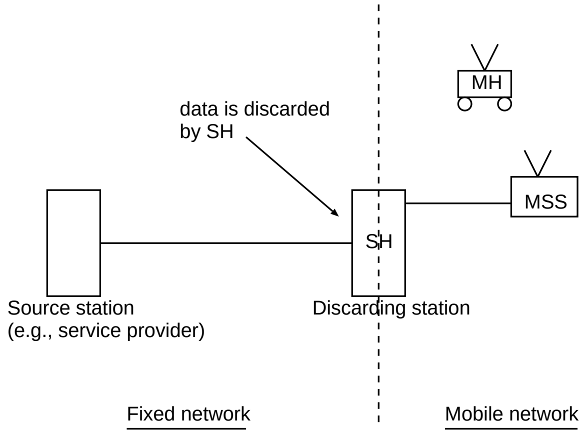


Figure 7: SH serves as a discarding station for connections to mobile hosts.

3.3 Algorithms to Discard Data

The SH needs to be instructed about *how* to discard data for a connection – i.e., clustered loss, uniform random loss or some other loss profile. Clearly one could define and implement a set of possible loss profiles that potentially cover all possible cases. However, such solutions that attempt to predict the nature of future applications and their preferred loss profiles would be risking obsolescence. As new applications are developed, the existing discarding mechanism(s) may prove quite useless or require significant modification, etc.

Our approach is to allow each application define its own loss profile. This may be done in two ways. One way for the application inform the SH about a suitable loss profile is to embed a *losstable* within the data stream informing the SH which segments to discard for different loss percentages. For example – for a 20% loss discard segments 1, 5, and 7 every x seconds, for a 40% loss discard 2, 3, 5 and 8 every x seconds, etc. Unfortunately this approach is very cumbersome and does not scale to arbitrary loss percentages.

We propose a different approach to the problem. The application user at the source station is provided with a *library* of "discarding" functions of which it chooses the most appropriate one that meets the demands of that application best. Each function is essentially a *discarding algorithm* which discards data in a certain specific manner. When the bandwidth is exceeded, the discarding station (i.e., the SH) executes the discarding function to discard segments from the data stream. For example, a certain function may discard segments based on a standard uniform distribution. Another may have the logic for discarding segments in clustered batches within a designated time interval, while another could discard segments at random. Indeed, with such a mechanism in place, it is quite possible to implement the deterministic case as

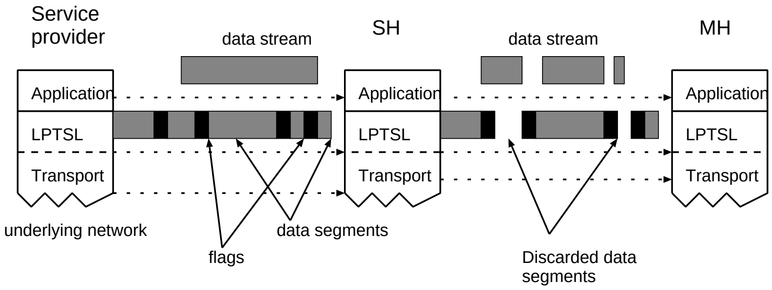


Figure 8: Functionality of LPTSL.

well. The possibilities are endless and the user is provided with the functionality of adding new functions as new applications are introduced to the system.

This approach also is attractive because, as discussed in Lazowska[17], a user linkable library provides for easier management. Thus, in the future, if a certain application cannot find a suitable discarding function amongst the existing ones within the library, a new one can be written and added to the library.

3.4 Sample Discarding Algorithms

We implemented two sample discarding functions – *clustered loss* and *random uniform loss*. Both functions accept the *discard_period* as a function parameter. It specifies in milliseconds, a certain time interval within which the discarding server attempts to discard segments to match up to the channel loss. If the computed loss is X%, and the discard_period is 'Y' msec long, then the server attempts to discard X% of the data in every 'Y' msecs. If it fails to discard the exact amount, it compensates for the difference during the next discard_period, and so on. On an average we meet our goal.

To verify the correct operation of our implementation, we applied the two discarding functions to two sample data streams. We considered an audio stream and a video stream. The audio stream consists of talk "spurts" which are interleaved with silence periods. The length of the talk spurts is exponentially distributed with a mean of 1.2 seconds while the length of the silence period is exponentially distributed with a mean of 1.8 seconds (see Brady[2]). The talk spurts generate fixed-sized segments of 160 bytes every 20 msec.

We generated a simulated MPEG-2 encoded video bit stream. The reader is referred to ISO 2-13818[24] for details on the MPEG-2 standard. We consider frames of 280 x 340 pixels in size (3 bytes per pixel) captured at 20 frames/sec. A group_of_pictures (GOP) consists of 12 frames and we assume a 50:1 compression ratio for a GOP. The average compressed I-type picture is about 11 KBytes and the remaining (P and B type) frames range in size between 5 and 11 KBytes.

Using the two discarding functions described for the audio and video simulations, we achieved results as shown in Table 2.

Figure 9 plots the total loss every discard period (in this case the discard period is 1000 msec) for the clustered loss algorithm for both types of data streams. The graph for video data shows a "band" around the 30% loss line. This happens because we cannot exactly match the 30% loss requirement during every discard period. As a result if the loss in the previous period was less than 30%, the loss in the current period is increased to compensate. This causes the wavy behavior. Audio clustered loss varies less because unlike video, audio segments are all of the same small size and thus we can match the loss percentage more precisely.

In Figure 10 we plot total loss at the end of a discard period (1000 msec) on the y-axis for all discard periods in the case of random uniform loss. Observe that the video loss (and audio loss) varies much more here than in the case of clustered loss. This is because we discard segments within a discard period

Application	**Loss Profile**	**Desired loss %**	**discard_ period**	**Actual loss % obtained**
audio	clustered	5	1000	4.997827
audio	clustered	5	2000	4.997827
audio	clustered	10	1000	9.952195
audio	clustered	10	2000	9.995654
audio	clustered	20	1000	19.904389
audio	clustered	20	2000	19.991308
audio	clustered	30	1000	29.856584
audio	clustered	30	2000	29.986962
audio	random	5	1000	4.823990
audio	random	5	2000	4.823990
audio	random	10	1000	9.734898
audio	random	10	2000	9.734898
audio	random	20	1000	20.643199
audio	random	20	2000	20.643199
audio	random	30	1000	30.421556
audio	random	30	2000	30.421556
video	clustered	5	1000	5.022101
video	clustered	5	2000	5.005231
video	clustered	10	1000	10.002529
video	clustered	10	2000	10.010876
video	clustered	20	1000	20.017451
video	clustered	20	2000	20.003087
video	clustered	30	1000	30.023991
video	clustered	30	2000	30.011005
video	random	5	1000	4.999510
video	random	5	2000	5.009099
video	random	10	1000	10.000834
video	random	10	2000	10.015903
video	random	20	1000	19.992799
video	random	20	2000	20.002819
video	random	30	1000	29.970323
video	random	30	2000	29.996829

Table 2: Experimental Results

randomly. However, at the start of the next discard period, we calculate the loss achieved during the previous period. If the loss was smaller than 30%, we calculate a new loss percentage for this discard period so as to compensate. This causes losses in consecutive discard periods to vary significantly.

The discarding algorithm attempts to meet the required loss of 30% by discarding a certain number of segments. However, it is unlikely that the actual loss within each discard_period matches 30% exactly. If the algorithm discards more than the required amount during a certain interval, it makes up during the next interval by discarding less than 30%. The results were encouraging and shows that the approach indeed works with real examples.

4 Implementation of LPTSL in High-Speed Networks

In developing our protocol we consciously attempted to reduce any inefficiencies. Research has indicated that performance bottlenecks are shifting from the network infrastructure to the transport system infrastructure resulting in a *throughput preservation problem*, where only a limited fraction of the available bandwidth is actually delivered to applications (see Schmidt[12]). One of the significant contributors to this problem is the excessive memory-to-memory copying that takes place amongst the various layers within a networking protocol suite. For example, in BSD UNIX, a piece of data is read and written at least twice in its entirety while traversing from the system call down to the network interface hardware (see Partridge[11]). To alleviate this problem we adopt the approach of Schwartz[8] where data is maintained in kernel space until transmission time.

The rest of this section is divided into three parts. In section 4.1 we discuss the implementation of LPTSL on the Multi-Stream Protocol of Schwartz[8] as an illustration. A similar description can be constructed for any other transport layer specification. For instance, we are currently building LPTSL on top of UDP (in the process UDP has been modified to make it more efficient). We discuss LPTSL header information embedded in the data streams in section 4.3.

4.1 LPTSL in the MultiStream Protocol

The MultiStream Protocol, MSP, is a feature-rich, highly flexible transport protocol, capable of being specified as a set of functions which the user enables or disables as required by the application or interconnecting networks. The user may make MSP "lean" by enabling only the most efficient functions required for protocol services while disabling unnecessary functions and services already supplied by the network and lower layers. This prevents unnecessary protocol processing overhead and improves overall performance of the network. Further, by dividing the protocol into minimally dependent parallel machines, MSP can achieve high performance while retaining its highly functional nature. The reader is referred to Schwartz[8] for details regarding the MultiStream protocol.

4.1.1 Modifications required to incorporate Loss Profiles

To modify the existing protocol to fit our operating needs in a mobile environment, we need additional functionality at the transport system in the form of new system calls and the ability to interpret modified packet formats. In order to achieve such functionality, we assume LPTSL logically resides right above the existing MSP transport layer. The application data is passed onto the LPTSL which performs encapsulation and then passes its output to the transport layer.

MSP offers its application users the *opena* and *openp* call to open transport connection endpoints. Each function returns a channel descriptor (e.g. a socket descriptor in BSD socket networking interface) which is then used by either end to communicate. Applications using LPTSL open connections in an identical way using the open command:

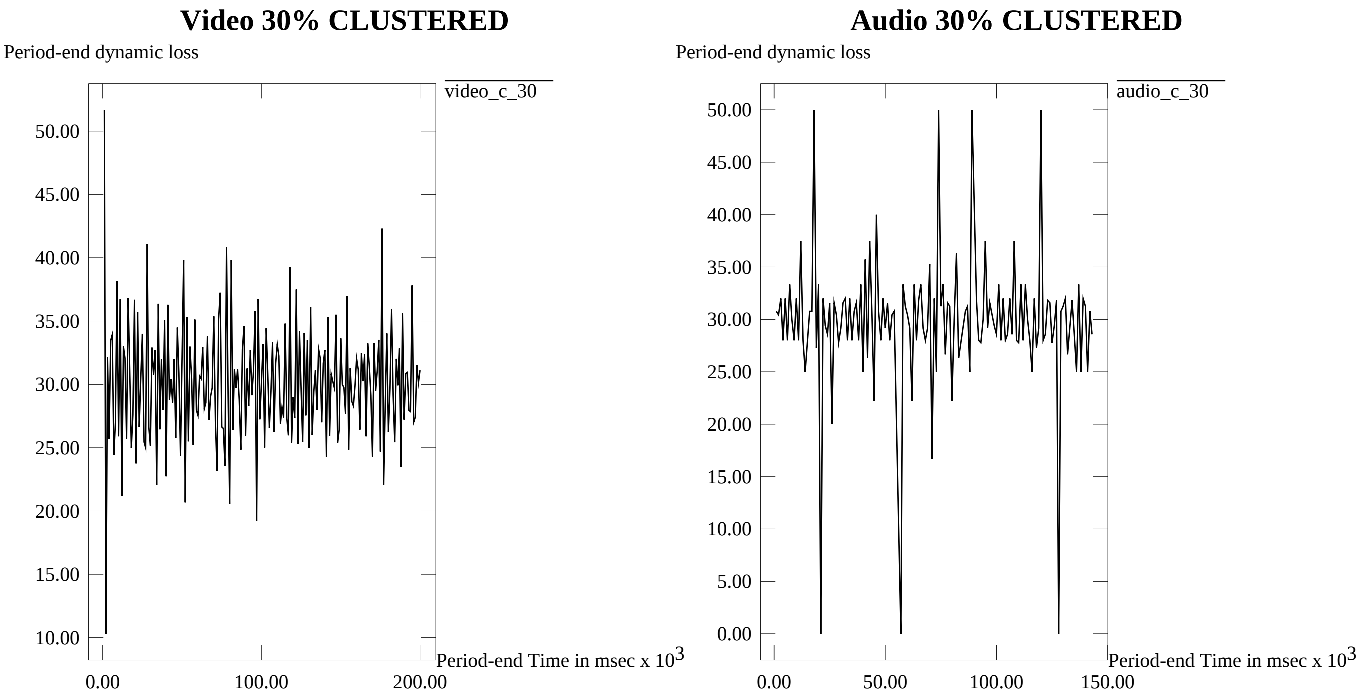


Figure 9: Overall clustered loss of 30%.

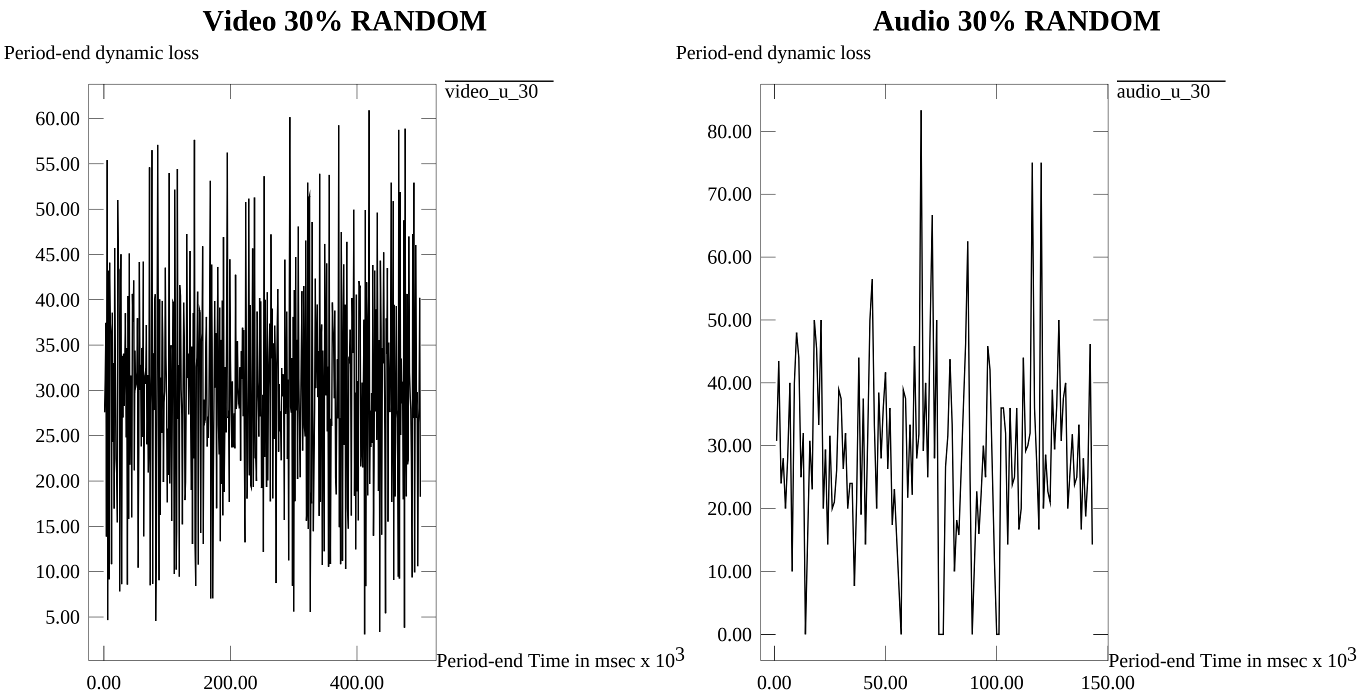


Figure 10: Overall uniform random loss of 30%.

int **lptsl_opena**(char * *segment*, ...);

int **lptsl_openp**(char * *segment*, ...);

The call returns a channel descriptor (e.g., a socket descriptor in BSD) and a connection identifier, *conn_id*. The other parameters are identical to the parameters in MSP calls. LPTSL performs the opena or openp calls on behalf of the calling program. A modification required to MSP is to ensure that all connections opened via LPTSL get tagged so that arriving data is passed to the application via LPTSL. Connections not using LPTSL have no need to interact with LPTSL at all.

Ordinarily, the appropriate "send" system call takes a pointer to a user defined data buffer which has been allocated in user memory space. The send call would then copy the data from the user buffer to a system buffer, held in kernel memory (the transport system). Instead, by supplying the user with a pointer to a kernel allocated data structure (buffer), we can skip over the copy process altogether. Note that it is possible to allocate such a buffer in a protocol such as MSP owing to its known, fixed bound on each packet's size (65K).

The first parameter in the open system calls given above is a character pointer that serves this purpose. In each of the open calls, the application user passes a character pointer that is initialized to NULL. The kernel detects an attempted connection establishment and after allocating a buffer in kernel memory, passes the appropriate[3] pointer back to the user.

MSP provides the *sndp* system call for the actual transfer of data. It takes a pointer to a buffer and passes the data on to the transport layer which generates a data packet. Each *sndp* call generates a transport packet that is then transmitted to the destination.

The purpose of LPTSL is to divide the user data into several segments, some of which may be discarded at a later stage. It is therefore necessary to provide a system call that lets LPTSL process the data and insert certain markers[4] within the data stream. The following system call provides that functionality:

void **lptsl_writeseg**(int *conn_id*, char * *segment*, int *priority*);

The first parameter, *conn_id* is the connection identifier, the result of the open calls described above. The character pointer, *segment,* is the one that had been passed to the user through the open call. Each *writeseg* call results in the kernel updating the character pointer (segment) to reflect the offset where the next write should take place. See Figure 11.

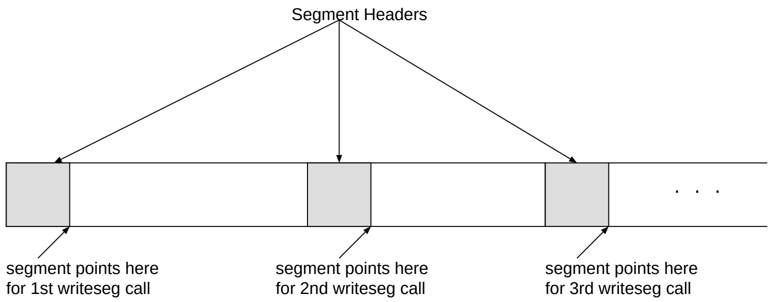


Figure 11: Writeseg system call

The application user makes successive writeseg calls, each time passing a certain amount of data that can semantically be considered as an integral segment of data. As an example, an encoded stream of video data would consider each frame as an integral segment. Each time the user calls writeseg, it passes a frame of data to the LPTSL. With each call, LPTSL prepends the data with its headers (encapsulation) and passes the new updated buffer pointer back to the user.

[3]Actually, the pointer passed back to the user is offset by a certain value to accommodate some header information.

[4]If the user data stream is logically sub-divided into several segments, the encapsulation headers would appear as *markers* within the data stream.

The third parameter, *priority* denotes the priority of the segment that is being passed. For an MPEG coded video stream, for example, the loss of certain frames (I type) is more significant than other types (P or B types). Therefore, the segment containing an I-type frame ought to have a higher priority associated with it.

void **lptsl_send**(int *conn_id*);

When the kernel buffer builds up to its maximum size (65K in MSP), or if the user makes an explicit *send* call, the LPTSL forces a *sndp*[5] call, passing its data to the MSP transport layer. The MSP transport system interprets the segments as user data and proceeds to build its own transport packets. Thus, with the functionality provided by these two additional system calls, the user never needs to make an explicit *sndp* call. In fact, it should now be masked off from the user.

4.2 Specifying the Loss Profile

Depending on the nature of the application, the user may select a certain *Loss Profile* for the data that may be discarded. Since there is no way of perceiving the nature of each and every type of application that would be used in future in the mobile environment, we have come up with a "scalable" solution to this problem. The user is provided with a library of several functions and chooses one that best suits his application.

void **lpfn**(int *conn_id*, int *ch_id*, char * *fname*, char * *fmtstr*, ...);

The system call, *lpfn* is used to inform the LPTSL about the user's choice of the discarding function, *fname*.

There may be some applications that would require a feedback from the discarding station in terms of bandwidth availability and the amount of loss that its application is currently suffering. More sensitive applications may utilize this information to possibly select a more appropriate discarding function than the one specified initially. The second parameter, *ch_id*, short for channel_identifier, provides the application a channel to listen upon. e.g. A UNIX user would, typically, open a socket for reading and pass its descriptor as the ch_id[6]. For some applications that do not care about changing conditions of the channel, a *ch_id* value of -1 indicates that the application does not want such functionality. In this case, LPTSL at the discarding station does not generate any channel_condition control packets.

The third argument, *fname* points to a character string which gives the name of the discarding function itself. The fourth parameter, *fmtstr*, is a format string for the variable list of parameters that specifies the data-type of each parameter that is passed to the discarding function. It is much the same as the format string specification in a "printf" or "scanf" call in C. The ellipses in the above function call denotes the variable list of parameters that are to be supplied to the discarding function.

The application user selects a particular function by its name, represented as a string of characters. To limit overhead costs to a minimum, LPTSL makes an internal system function call to obtain an index value corresponding to the specified function name.

int **fnindex**(char * *function_name*);

The function is implemented as a single dimensional array of character pointers that list each of the discarding functions that are currently defined in the library. When a call is made, a simple lookup yields an index corresponding to a name.

Although the discarding function is selected at the source station, the actual process of discarding segments takes place at an intermediate discarding station (the SH). The function index along with its

[5]For MSP block streams, the LPTSL would make a *sndb* call instead.

[6]One possible implementation is as following. The application starts a process initially and `open`s a UNIX socket for reading. The returned value of the socket descriptor is assigned to ch_id. It then `fork`s a child process which in turn `exec`s another process that makes the *writeseg* calls. The parent process, meanwhile, performs a blocking `read` on the socket ch_id, waiting for any channel_condition control packets (see section 4.4) that may arrive on this socket.

optional set of parameters are therefore sent over the network to the discarding station. LPTSL then obtains the function name corresponding to the index that was received. There is however, no guarantee that the function name just obtained is exactly the same as the user had originally specified. Indeed, the library at the discarding server could be a different version from the one at the source station. To prevent such a thing from happening, a special field called the *library_version* number is included along with the function details.

int **lpfnlibver**();

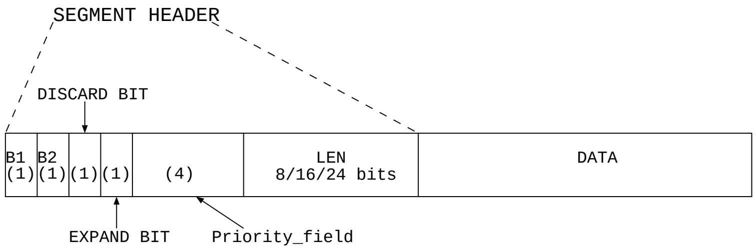
The LPTSL at the source station makes a call to *lpfnlibver* to obtain the library_version number at his end. When the discarding station receives this information, it makes a local call to its library and verifies whether the two version numbers match. If the match fails, the process is aborted.

void **listlpfns**();

This is a utility function that a user may call to obtain a list of all the functions that are currently available in the library. A brief description is included along with each function's name.

4.3 LPTSL packetization

The LPTSL prepends a segment header to each data packet that is passed to it through the writeseg calls. The size of the header can vary from 1 byte up to 3 bytes. The header format is shown in Figure 12.



B1	B2	LEN Bits Used	Max Segment Size	Segment Header Size
0	0	8	256 Bytes	2 Bytes
0	1	16	64 KB	3 Bytes
1	0	24	16 MB	4 Bytes

Figure 12: Structure of a data segment

The first two bits are the *size_decoding* bits (B1, B2 in the figure). The third bit is called the *discard bit* and the fourth bit is called the *expand bit.* The next 4 bits are the *priority_field* bits and are used for encoding the priority value of the data segment. The 4 bits provide upto 16 different priority levels that can be assigned to each segment.

The size of each segment header is determined by examining the first 2 bits of the header. The two bits allow four values, one of which (binary 11), denotes a LPTSL control packet and is explained in section 4.4. This first three values (binary 00, 01, 10) gives us three possible values for the length of the segment header - 2, 3 and 4 bytes respectively (see Figure 12).

For each of the three cases, with the first 8 bits being used up, the remaining bits yield maximum segment lengths of 256 bytes, 64K bytes and 16 MBytes respectively.

This flexibility allows the LPTSL to choose the most appropriate size for the segment header during the encapsulation process at each writeseg call. The resulting data stream may consist of segments of largely varying sizes, but each would have its appropriate segment header attached to itself.

The LPTSL at the decoding station simply examines the first two bits and extracts the length of each segment followed by the data itself.

Initially, the source machine's LPTSL sets the *discard_bit* to 0 for each segment. At the discarding station, as the data segments are regenerated and passed up to the LPTSL, it sets the discard_bit to 1 for each segment that it discards. Otherwise, the bit remains zero, indicating that the corresponding segment data follows in its entirety. It is important to note that the *discard_bit* is set to 1 only once for each discarded segment and this can only be done at the station that is discarding the data. Although the data within a segment may be discarded, the segment headers are always left intact.

Figure 13 gives an example of a resulting output data stream.

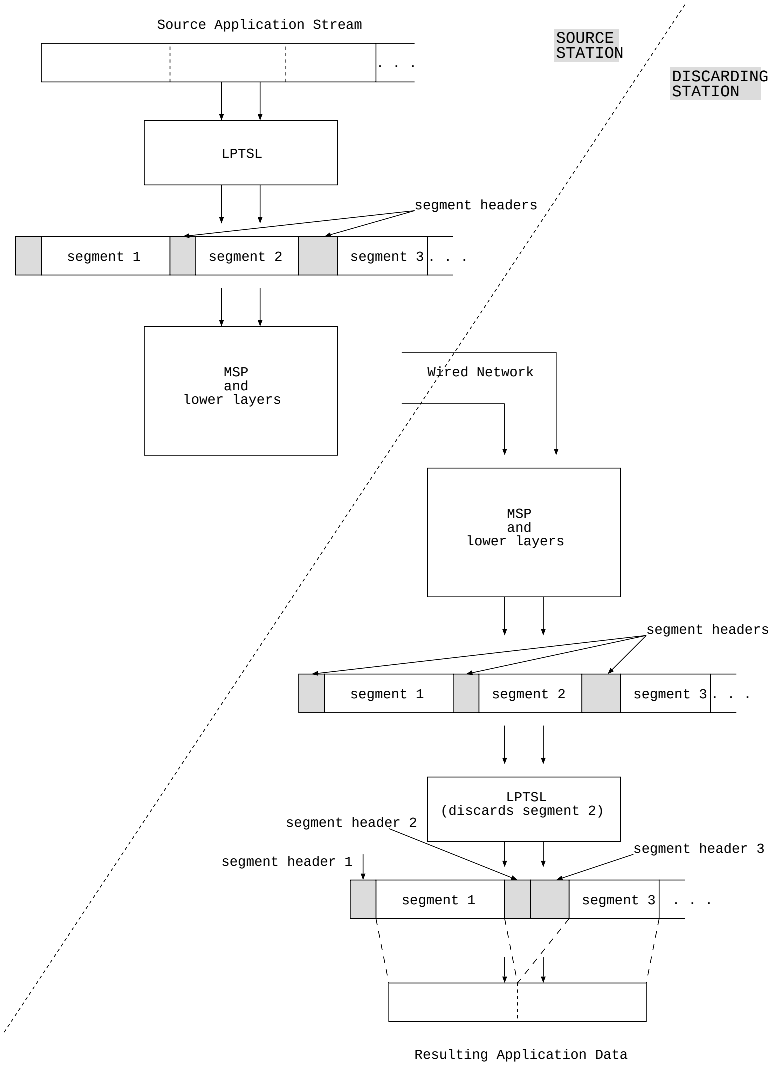


Figure 13: Resulting output data stream

The *expand_bit* is used to indicate to the final destination's LPTSL as to whether the discarded segments are to be re-expanded to NULL bytes. When set to 1, the LPTSL at the destination machine reads the length from the segment header and generates the proper number of NULL bytes. When the expand_bit is set to 0, no such expansion takes place at the final destination.

This functionality may be required for some applications that rely on synchronized reception of data at the final destination's application layer. e.g. A particular application may consist of an encoded bit stream to be transmitted in a synchronized manner, with the requirement that the incoming bit-rate as well as the exact number of bytes being received be exactly the same as the original source application at the transmitting station.

4.4 LPTSL control packets

As explained in section 4.3, when the first two bits of a segment header are set to binary "11", it is treated as a LPTSL control packet[7]. For control packets, the third and fourth bits constitute the *type_id* field, and it identifies different types of control packets. Refer to Figure 14.

The discarding function, its associated list of parameters and an optional channel_identifier is provided by the user on the source machine. However, since the actual process of discarding data takes place on another machine (refer figure 13), we need to send all this information to the discarding station which would then execute the specified discarding function. However, sending the entire string of characters is not the most efficient way for transmitting over the network. Instead, LPTSL, at the source station makes an internal call[8]to the function library and obtains the function index corresponding to that function name.

Since we do not envision more than 256 different discarding functions, simply using 1 byte suffices as a *function_index*. In addition to this, we also need to send the *library_version* number from the source station to the discarding station. As explained in section 4.2, we must ensure that the library functions on both stations are identical. For this purpose, the 1 byte *library_version*[9] library_version number is sent along with the function index and its associated parameter list. Together, they constitute a *lp_function* packet. See figure 14.

The parameters that are used by the discarding function are encoded in the control packet as follows. The function_index is followed by a 1 byte field which gives the number of arguments that follow. Each argument is preceded by a 1-byte length field which gives the number of bytes used to represent it. e.g, a short integer parameter has a length field equal to 2, followed by the actual value represented in two bytes.

The format string, *fmtstr* indicates the exact number and the type (and hence their length in bytes) of each parameter.

As indicated in figure 14, the different types of control packets are distinguished by a 2 bit *type_id* field. A value of "00" indicates a *lp_function*, while a *channel_condition* control packet is indicated with a type_id of "10".

The *channel_condition* control packet may be sent from the discarding station to the source station to inform about changing conditions of the communication channel. Such a packet is generated only if the user had specified a *ch_id* value other than -1.

The *b/w_loss* field is a two byte short integer reflecting the actual percentage of loss that the particular application is currently suffering at the discarding station. e.g. If the available bandwidth has been reduced to 90% of its original value, the b/w_loss would have a value of 10.

The *segment_discard_time* field is a two byte short integer specifying the *discard_period* in milliseconds. Refer to section 3.4 for details.

The third type of control packet is the *abort* packet and is identified by a *type_id* field set to "11". It is generated by the LPTSL at the discarding station in case the *library_version*s fail to match up at the discarding station. On receiving this packet, the LPTSL at the source station informs the user and aborts the process. This packet does not contain any other information.

[7] Note that it is referred to as a control *packet*, as opposed to a control segment. The term packet is used when referring to smaller units of information, as opposed to entire data segments which may be quite large.

[8] refer to function call *fnindex()*.

[9] refer to the *lpfnlibver()* system call.

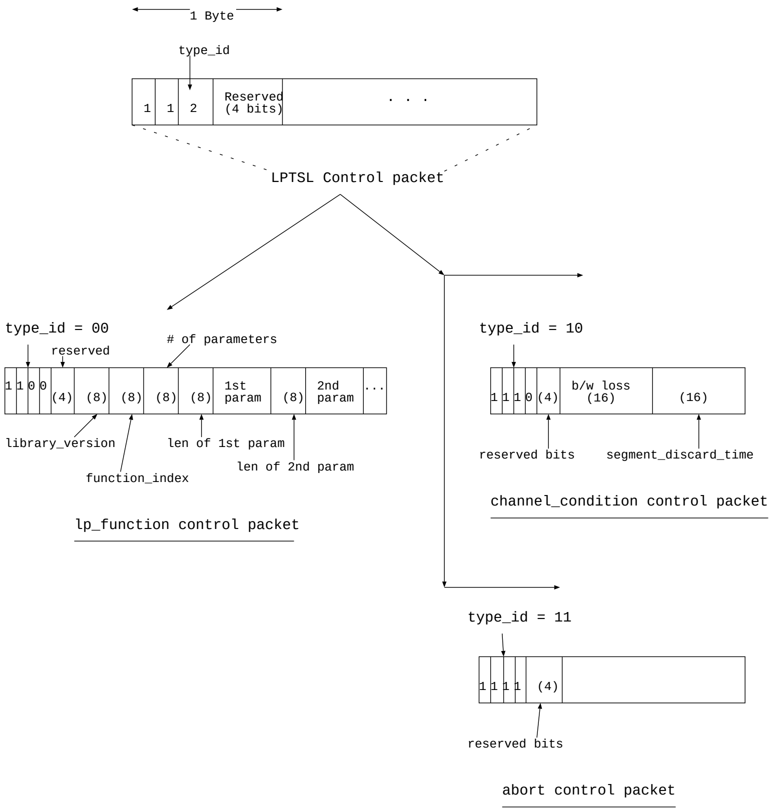


Figure 14: LPTSL control packets

5 LPTSL in the MPEG-2 Transport System

A video elementary stream consists of I, P, B pictures known as presentation units as per the MPEG-2 terminology. The presentation units, when encoded and prepared for transmission are called access units. Figure 15 describes the decoding sequence for a MPEG video clip.

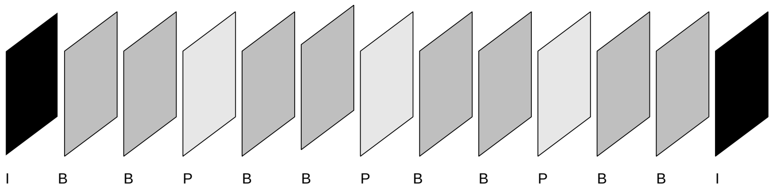


Figure 15: Decoding order of MPEG-2 video segment

For video elementary bit streams, some of the access units may not be transmitted in presentation order. If an intra-coded (I) or a predictive-coded (P) picture is carried before one or more bidirectionally-coded (B) pictures, then it must be delayed in the target decoder's reorder buffer. While that frame is stored in the buffer, the subsequent B pictures are decoded and presented.

To implement loss profiles data needs to be discarded from such a video stream. Note that we need to discard entire pictures(I,P or B) as there is no point in discarding a portion of a picture, while retaining a part of it. The consequences of discarding I, P or B frames are, however, different.

Discarding a B picture does not prevent us from decoding and presenting the other pictures from the stream. Discarding a P type of picture however, is more costly. Not only is that picture lost, but even the subsequent (w.r.t the transmission order) B pictures, till the next P or I picture, are rendered useless. Without the P picture which is a reference for the bi-directionally interpolated image, the B picture(s) cannot be decoded. The worst case is when an I picture is discarded. The entire group of pictures up until the next I picture is received and decoded have to be thrown away.

In section 5.1 we summarize how the MPEG-2 transport system works. All of this discussion is based in [23, 24, 25]. Section 5.2 then discusses how LPTSL can be incorporated into MPEG-2 transport system.

5.1 Summary of MPEG-2 System Architecture

Video and audio data is compressed to form elementary bit streams which are packetized and grouped together with a common time base to constitute a program. These programs and an overall system control data stream are then asynchronously multiplexed together to form a MPEG-2 Transport Stream. The Transport System combines multiple programs with one or more independent time bases into a single stream.

User data from each elementary stream is encoded and multiplexed together with control information that allows the constituent elementary streams to be replayed in synchrony. Each elementary stream consists of access units, which are the coded representation of presentation units. For a video elementary stream, the presentation unit is a picture while for audio it would be an audio frame. In addition to the user data carried in various elementary streams, the Transport Stream carries certain information in the form of Program Specific Information (PSI) tables that identify each of the elementary bit streams for the purpose of decoding and presentation.

Each of the elementary streams is packetized into variable length *Packetized Elementary Stream (PES)* packets. A PES packet consists of a packet header followed by user data. The PES packets are then inserted into fixed sized Transport Packets of 188 bytes in length. The first byte of each PES packet header is located at the first available payload location of a Transport Packet. This implies that the last Transport packet corresponding to a PES packet may only be partially filled up with user data. Stuffing bytes are then used to bring the packet up to its required size of 188 bytes.

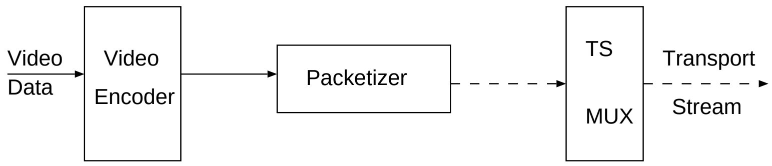


Figure 16: Encoding a video bit stream

Transport Packets begin with a 4 byte prefix which contains a 13 bit Packet Identifier (PID) field. Via the PSI tables, the PID identify the contents of the data in the Transport Packet. Transport packets of one PID value carry data of one and only one elementary stream.

5.1.1 Program Specific Information

There are four types of PSI tables that are carried in a MPEG-2 Transport Stream:

- **Network Information Table**
- **Conditional Access Table**
- The **Program Map Table** carries information about the applications that make up programs. It provides the mappings between program numbers and the elementary bit streams that comprise

them. The program_map_table is a complete collection of all program definitions for a Transport Stream. The table itself is broken up and transmitted in packets, the PID values of which may be privately selected. It contains a 13-bit PCR_PID indicating the PID of the Transport Packet which contains the PCR of the program specified by the corresponding program number.

The listing given below indicates the various fields along with the number of bits used to represent each field. Note the fields that have a multipler 'N'[10].

program_map_table	
table_id	8
section_syntax_indicator	1
'0'	1
reserved	2
section_length	12
program_number	16
reserved	2
version_number	5
current_next_indicator	1
section_number	8
last_section_number	8
reserved	3
PCR_PID	13
reserved	4
program_info_length	12
descriptor	variable
stream_type	8 * N
reserved	3 * N
elementary_PID	13 * N
reserved	4 * N
ES_info_length	12 * N
descriptor	variable * N
CRC_32	32

program_association_table	
table_id	8
section_syntax_indicator	1
'0'	1
reserved	2
section_length	12
transport_stream_id	16
reserved	2
version_number	5
current_next_indicator	1
section_number	8
last_section_number	8
program_number	16 * N
reserved	3 * N
network_PID — program_map_PID	13 * N
CRC_32	32

- The **Program Association Table** describes how program numbers are associated with programs and contains the 13-bit PID of the Transport Packets which contain the program_map_table for these programs. The table is itself carried in Transport Packets having a PID = 0.

Figure 17 illustrates the entire process of extracting elementary bit streams for a program at the receiver.

5.1.2 Transport Packet

The following is a table of each of the fields in a Transport Packet. The number at the end of each field indicates the number of bits used to represent that field. The sync_byte is a fixed 8 bit field which marks the beginning of a transport packet. Its value is 0x47. The adaptation_field_control indicates whether this transport packet contains an adaptation field and/or payload. The data_bytes are contiguous bytes from

[10]The *program_map_table* is the complete collection of all programs that make up a transport stream. The multiplying factor indicates the 'N' elementary streams whose descriptions have been concatenated together to form a "program definition".

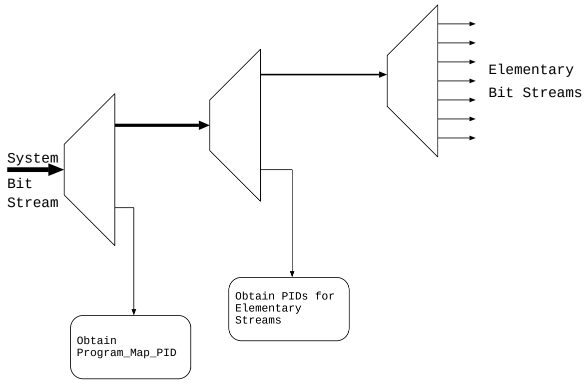


Figure 17: System Level Demultiplexing

the PES data stream, PSI tables or private data as indicated by the associated PID field. The presence of the adaptation field is indicated by the adaptation_field_control flags. In the adaptation field, the PCR field is used for synchronization purposes and is explained in a later section. The stuffing_byte field is inserted by the encoder and quietly discarded by the decoder.

transport_packet	
sync_byte	8
transport_error_indicator	1
payload_unit_start_indicator	1
transport_priority	1
PID	13
transport_scrambling_control	2
adaptation_field_control	2
continuity_counter	4
adaptation_field	conditional
data_byte	variable

adaptation_field
adaptation_field_length, 8
discontinuity_indicator, 1
random_access_indicator, 1
elementary_stream_priority_indicator, 1
PCR_flag, 1
. OPCR_flag, 1
splicing_point_flag, 1
transport_private_data_flag, 1
adaptation_field_extension_flag, 1
program_clock_reference (PCR), 42
original_program_clock_reference (OPCR), 42
splice_countdown, 8
transport_private_data_length, 8
adaptation_field_extension_length, 8
private_data_byte, 8
stuffing_byte, 8 * N

5.1.3 PES Packet

The variable sized PES packet has a number of fields that are used for various purposes. They are listed below:

PES packet

- **packet_start_code_prefix, 24**. Combined with the stream_id that follows it constitutes a packet start code that marks the start of a PES packet. Its value is set to 0x000001.
- **stream_id, 8** specifies the type and number of the elementary stream. Refer to Table 3 for details.
- **PES_packet_length, 16**
- **PES_scrambling_control, 2**

- **PES_priority, 1**
- **data_alignment_indicator, 1**
- **copyright, 1**
- **original_or_copy, 1**
- **PTS_DTS_flags, 2** indicate the presence of the Presentation and/or Decoding Time Stamps in the packet.
- **ESCR_flag, 1**
- **ES_rate_flag, 1**
- **DSM_trick_mode_flag, 1**
- **additional_copy_info_flag, 1**
- **PES_CRC_flag, 1**
- **PES_extension_flag, 1**
- **PES_header_data_length, 8**
- **PTS (presentation_time_stamp), 33.** Coded in three separate fields, it indicates the intended time of presentation in the system target decoder of the presentation unit that corresponds to the first access unit that lies in the packet. For a video PES, an access unit commences in the packet if the first byte of a video picture_start_code is present.
- **DTS (decoding_time_stamp, 33.** Again, coded in three separate fields, it indicates the intended time of decoding of the first access unit in the packet.
- **ESCR, 42**
- **ES_rate, 22**
- **trick_mode_control, 3**
- **additional_copy_info, 7**
- **previous_PES_packet_CRC, 16**
- **PES_private_data_flag, 1**
- **pack_header_field_flag, 1**
- **program_packet_sequence_counter_flag, 1**
- **P-STD_buffer_flag, 1**
- **PES_extension_field_flag, 1**
- **PES_private_data, 16**
- **pack_field_length, 16**

stream_id	stream_type
1011 1100	program stream map
1011 1101	private_stream_1
1011 1110	padding stream
1011 1111	private_stream_2
110x xxxx	MPEG audio stream - number xxxxx
1110 xxxx	MPEG video stream - number xxxx
1111 0000	ECM
1111 0001	EMM
1111 0010	DSM CC
1111 0011	MHEG (ISO 13522)
1111 xxxx	reserved data stream - number xxxx
1111 1111	program stream directory
The notation x means that the values 0 and 1 are both permitted and result in the same stream type. The stream number is given by the values taken by the x's.	

Table 3: MPEG-2 Elementary stream_id's

- **program_packet_sequence_counter, 8**
- **original_stuff_length, 7**
- **stuffing_byte, 8 * N**
- **P-STD_buffer_scale, 1**
- **P-STD_buffer_size, 13**
- **PES_extension_field_length, 7**
- **PES_packet_data_byte, 8 * N.** These are contiguoes bytes from the elementary bit stream as indicated by the packet's stream_id. The number of packet_data_bytes, N, can be calculated from the PES_packet_length_field.
- **padding_byte, variable**

Figure 18 describes the structure of a PES packet in detail. The user data carried within the PES data fields depends on the type of application. The listing below illustrates the hierarchical coding structure of a MPEG-2 video stream. The number at the end of each field indicates the size of the field in bits.

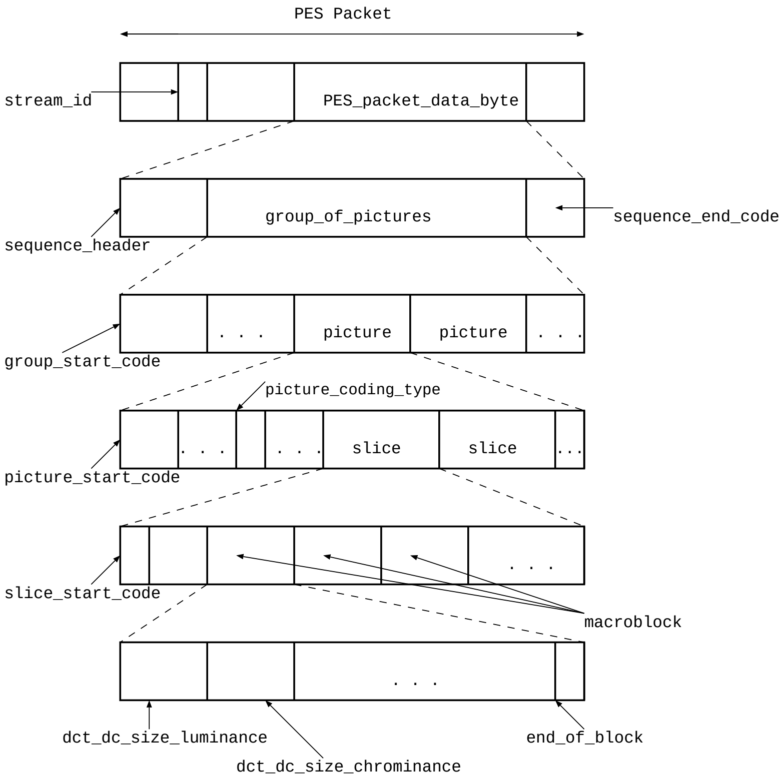


Figure 18: Structure of a video PES packet

group_of_pictures	
group_start_code	32
time_code	25
closed_gop	1
broken_link	1
extension_start_code	32
group_extension_data	8
user_data_start_code	32
user_data	8
picture	variable

picture	
picture_start_code	32
temporal_reference	10
picture_coding_type	3
vbv_delay	16
full_pel_forward_vector	1
forward_f_code	3
full_pel_backward_vector	1
backward_f_code	3
extra_bit_picture	1
extra_information_picture	8
extension_start_code	32
picture_extension_data	8
user_data_start_code	32
user_data	8
slice	variable

The picture_coding_type identifies the coding type of the picture - intra-coded (I), predictive-coded (P), bidirectionally predictive-coded picture (B) or intra-coded with DC coefficients (D) as illustrated in Table 4, see [24].

picture_coding_type	coding method
000	forbidden
001	intra-coded (I)
010	predictive-coded (P)
011	bidirectionally-predictive-coded (B)
100	dc intra-coded (D)
101	reserved
...	...
111	reserved

Table 4: MPEG-2 video stream picture codes table

5.1.4 Decoding

A decoding system deals with reconstruction of presentation units from compressed data and presents them in a synchronized sequence at the proper presentation times. Since received data is expected to be processed at a particular rate (the rate at which it was generated and transmitted), loss of synchronization leads to total loss of presentation/display synchronization.

Since a digital compressed bit stream carries variable amounts of picture information per frame (based on coding approach and complexity), the synchronization timing cannot be derived directly from the start of picture data. A digital bit stream has no concept of synch pulses that are present in the traditional television channel formats like NTSC.

The solution to this issue is to transmit timing information in the adaptation headers of selected packets, to serve as a reference for timing comparison at the target decoder (T-STD). This is achieved by transmitting a sample of a 27 Mhz clock in the *program_clock_reference (PCR)* field. The time sample indicates the expected time at the completion of reading of that field from the bit stream by the transport decoder. With this information in place, the audio and video sample clocks in the decoder (T-STD) are locked to the system clock derived from the PCR values.

The Transport Stream System Target Decoder (T-STD) takes an MPEG-2 Transport Stream as its input. Although a Transport Stream may contain multiple programs multiplexed together, the T-STD decodes only one program at a time. For the T-STD timing model, all indications refer to the particular time base of that program.

The input data from the MPEG-2 Transport Stream enters the decoding system at a piece-wise constant rate. The i^{th} byte, M(i), enters the decoder at time $t_m(i)$. This time is obtained by decoding the input PCR fields which are encoded in two parts, described as follows: One, is in units of 1/300 times the system clock frequency, called the PCR_base, and another, called the PCR_ext, in units of the system clock frequency itself. The value indicated by the PCR(i') field indicates time $t_m(i')$, where i' refers to the last byte of the PCR_base field, M(i').

The input arrival time, $t_m(i)$, of all other bytes is constructed from PCR(i') and the *transport_rate* at which data arrives. The *transport_rate* is computed as the number of bytes in the Transport Stream between the last bytes of two successive PCR fields of the same program, divided by the difference between the time values encoded in these same two PCR fields.

Specifically:

$$t_m(i) \;=\; \frac{PCR(i'')}{system\_clock\_frequency} + \frac{i - i''}{transport\_rate} \tag{1}$$

where:

i'' is the index of the final byte of the *program_clock_reference* base field in the transport packet adaptation field and

i is the index of any byte in the Transport Stream. [23]

It is important to note that if a PCR field is present in the adaptation field, that PCR field must be valid for the elementary stream contained in its transport packet.

5.2 Modifications to the MPEG-2 Transport System to support Loss Profiles

The MPEG-2 Transport System as described so far offers no support for implementing Loss Profiles for applications operating in lossy mobile environments. As in the case of MSP, we define a new **Loss Profile Transport Sub-Layer, LPTSL** that introduces functionality needed for supporting applications with user-specified loss profiles.

Although the MPEG-2 Transport System (ISO/IEC 13818 standard) is considered as an integral system responsible for the entire process from encoding to decoding and presentation of application data, the actual Transport System is essentially responsible for forming and transmitting MPEG-2 Transport packets only. The task of encoding and formation of packetized elementary bit streams and the corresponding decoding and presentation of the same is a higher level functionality that cannot be considered as part of the transport system [11].

Consequently, we define LPTSL as a transport sub-layer residing in between the PES packetizing layer and the transport layer which takes the various PES packets along with necessary system information and forms fixed length transport packets of 188 bytes in length.

At the source station, LPTSL defines segment boundaries within the incoming PES packets and outputs its resulting bit stream onto the existing MPEG-2 transport system. The transport system takes its (modified) input stream and packetizes it into transport packets. Figure 19 describes the system after incorporating the LPTSL into the existing transport system.

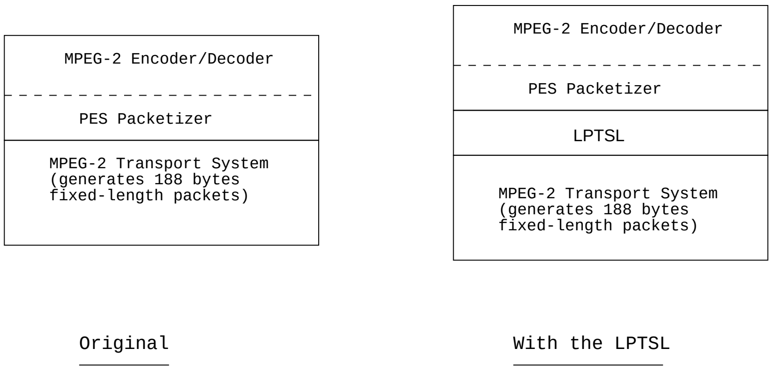


Figure 19: Incorporating the LPTSL in an MPEG-2 Transport System

5.2.1 Working of the LPTSL

The MPEG-2 Transport System has a tight timing model. The entire process of decoding and demultiplexing of the transport packets takes place in a synchronized manner with the associated timing information

[11] As shown in section 5.1.3, a PES packet contains timing information in the form of *PTS/DTS* fields that are dependent on the encoding process itself. It contains the *data_alignment_indicator* field which refers to the access units that are carried within the data portion of a PES packet. Clearly, the PES packetizing process is distinct from the functioning of the transport system.

being carried in the *PCR* and *OPCR* fields in the adaptation header of certain transport packets (refer to section 5.1.4).

After the LPTSL at the source station has modified the application bit stream, the transport layer simply uses the modified output bit stream and encapsulates it into fixed sized transport packets. It includes appropriate time stamps so that the destination transport system can decode and re-generate the original bit stream. Thus, as far as the receiving and decoding process is concerned, the entire process can take place in synchrony.

When the transport stream enters a discarding server station, the transport system recovers the bit stream as the source station had modified it and passes it onto the LPTSL. Depending on the channel bandwidth available to each application, the LPTSL discards segments of data based on the chosen loss profile.

Each segment of data has an associated header that gives the length of that segment. When bytes are to be discarded, only the data portion of each segment is thrown away, the segment headers are left intact.

The LPTSL then passes the resulting bit stream back to the transport system for its packetization. Transport packets are formed with PCR/OPCR fields reflecting the new timing information to be used at the transport decoder at the final destination.

Finally, when the transport packets are received at the destination they are synchronously decoded and the resulting bitstream is passed up to the LPTSL system. The destination application (the MPEG-2 decoder in this case), receives the partially discarded PES packets from the LPTSL and decodes them for presentation to the user. Section 5.2.3 describes the use of the LPTSL control *expand_bit* which enables the application decoder to function in a synchronized manner.

Figure 20 describes the functioning of the LPTSL. Note how the 4th segment is discarded by the intermediate discarding station and later expanded at the destination station. This is accomplished by the use of the *expand_bit* in the LPTSL segment header (explained in section 5.2.3).

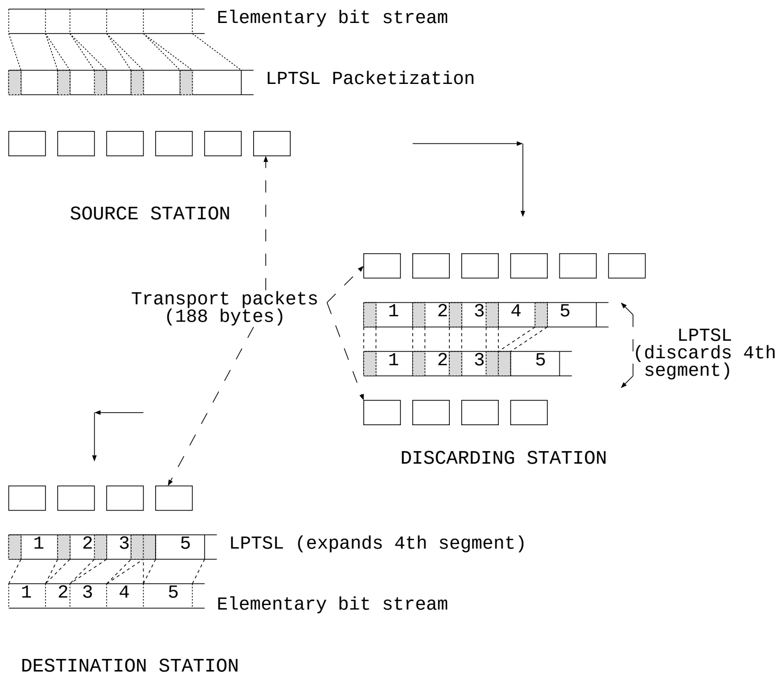


Figure 20: Functions of the LPTSL

5.2.2 System calls

For a particular application, it is the application user who knows about the type of application that is being used, e.g. audio or video. The transport system has no clue as to what constitutes the application data and hence has no way of marking out boundaries within the data stream. It is therefore imperative that the user at the source station make appropriate function calls to the LPTSL to mark out segments within its application data that may have some semantic meaning associated with themselves. e.g., For a video bit stream, each I/P/B-type encoded picture frame would constitute a segment.

At the source station, the input to the LPTSL is a data stream consisting of PES packets and PSI tables. Since PSI tables are necessary for the demultiplexing and decoding of the elementary bit streams, the LPTSL never considers these segments for discarding purposes. Even for PES packets, only the segments formed out of the data bytes are considered for discarding. The PES headers are never discarded.

int **lptsl_open**(char * *segment*, ...);

This system call refer to connection establishment. It returns the connection identifier which identifies the local endpoint of the connection. The parameter *segment* is a character pointer, initially set to NULL, that is passed to the kernel which allocates a kernel buffer and sets *segment* to point to it[12].

void **lptsl_writeseg**(int *conn_id*, char * *segment*, int *priority*);

As the encoder generates an output bit stream, it is passed to the PES packetizer which forms the PES packets along with the appropriate packet header. The application user, at the packetizing stage, makes an explicit *writeseg* call to the LPTSL to mark the end of a data segment. The LPTSL attaches a segment header and returns the updated pointer in *segment*. A *writeseg* call is used to mark out a segment of the application's data that has some semantic meaning associated with itself [13]. A segment would consist of either an I-type or P-type or a B-type encoded picture.

Figure 21 shows how the *segment* pointer gets updated each time a *writeseg* call is made.

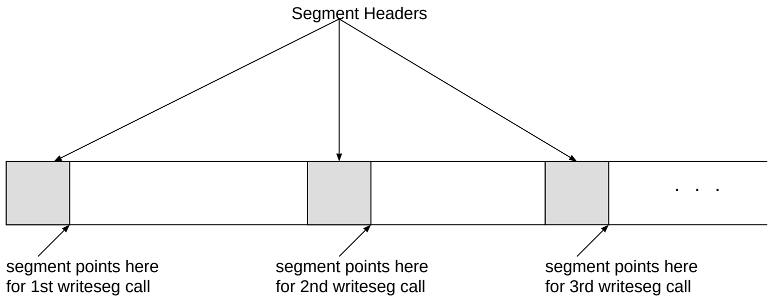


Figure 21: writeseg calls

The third parameter specifies the priority to be associated with the current segment. An MPEG-2 encoded video sequence is made up of three - I, P and B-types of picture frames. Note that the loss of a B-type picture is the least "expensive", followed by that of a P-type and then an I-type picture frame. The priority value could be set accordingly to indicate this. Apart for the video data itself, each PES packet's header as well as each of the PSI tables would form segments having a high priority value associated with themselves.

With each *writeseg* call, a new segment is formed with its associated segment header. The modified structure of a PES packet is listed below and it shows the LPTSL segment headers that are formed with each *writeseg* call.

[12]Note that this buffering is already being done at the PES packetizing layer (refer ISO 1-13818[23]). The buffer management scheme can be modified to incorporate the changes required to manage the *segment* pointer.

[13]e.g. a video PES consists of several *pictures* grouped together to form a *group_of_pictures* (see section 5.1.3)

PES packet

- **LPTSL segment header, 16/24/32** †
- packet_start_code_prefix, 24
- stream_id, 8
- PES_packet_length, 16

.
.
.

- P-STD_buffer_scale, 1
- P-STD_buffer_size, 13
- PES_extension_field_length, 7

$\boxed{\textbf{writeseg call}}$

- **LPTSL segment header, 16/24/32** ‡
- PES_packet_data_byte, 8 * N_{i_1}

$\boxed{\textbf{writeseg call}}$

- **LPTSL segment header, 16/24/32** ‡
- PES_packet_data_byte, 8 * N_{i_2}

$\boxed{\textbf{writeseg call}}$

.
.

- padding_byte, variable

Note the various segments that are formed as a result of multiple writeseg calls. The first *writeseg* call is made immediately after the PES packet header is generated. The LPTSL attaches the corresponding segment header at the beginning of each segment†(refer to section 5.2.3 for the header structure). The subsequent *writeseg* calls generate segments as shown above‡.

void **lptsl_send**(int *conn_id*);

When the kernel buffer builds up to its maximum allocated size (PES packets have a maximum size limit of 64K bytes), or if the user makes an explicit *send* call, the buffers are flushed out to the transport system. For the MPEG-2 system, this would be done at the end of each PES packet. In both the above system calls, the parameter *conn_id* refers to the connection identifier which was returned by the open system call.

5.2.3 LPTSL packetization

Each data segment has an associated segment header which can vary from 2 to 4 bytes in length. The format of a LPTSL data segment is given in the table below.

LPTSL data segment	
size_decoding_bits	2
discard_bit	1
expand_bit	1
priority_field	4
segment length	8/16/24
PES data	variable

The discarding station's LPTSL sets the *discard_bit* to 1 for each segment that gets discarded. Otherwise, it is set to 0.

The MPEG-2 application decoders rely on timing information that enables the decoding and presentation process to take place in a synchronized manner. In particular, the application encoders at the source machine generate PTS/DTS time stamps that are required for proper decoding and presentation of the application. These time stamps are calculated on the basis of the number of bytes that are "encountered" during the encoding process. If, however, some of these bytes are discarded at an intermediate station, the time stamps would reflect incorrectly on the resulting bit stream, throwing the entire process out of synchrony.

To prevent this from happening, the LPTSL at the discarding station sets the *expand_bit* of the segment header to 1 each time it discards a segment. The header contains the length of the discarded segment and the fact that its data has been discarded. Since the *expand_bit* also set to 1, when the decoder at the destination sees this segment, it "expands" the segment up to its original size by generating the requisite number of NULL bytes. The PTS/DTS time stamps that are contained in the PES headers can now be used to decode and present the access units to the user. The source encoder and the destination decoder are oblivious to the fact that data was discarded by some intermediate station. The *priority_field* is 4 bits long thus providing 16 different levels of priority that each segment can be assigned to.

5.2.4 Specifying the Loss Profile

As in the case of MSP, the user is provided with a library of several functions and chooses one that best suits her application. The function *listlpfns()* provides a list of all the discarding functions that are available in the library.

void **lpfn**(int *conn_id*, int *ch_id*, char * *fname*, char * *fmtstr*, ...);

The system call, *lpfn* is used to inform the LPTSL about the user's choice of the discarding function, *fname*. The first parameter, conn_id identifies the particular connection that is being referred to. The second parameter , *ch_id* is always set to a value of -1 for MPEG-2 systems.

The discarding function is indicated by *fname* and its associated argument list is denoted by the format string *fmtstr*. It is followed by a variable number of parameters that are to be used by the discarding function, *fname*. The ellipses denote the variable list of parameters that are to be supplied to the discarding function.

5.2.5 LPTSL control packets

When the first two bits of a segment header are set to binary "11", it is treated as a LPTSL control packet. The packet structure is shown in the table given below.

LPTSL control packet	
'11'	2
'0'	1
reserved	1
library_version	8
fnindex	8
number of parameters	8
length of 1st parameter	8
1st parameter	variable
length of 2nd parameter	8
2nd parameter	variable
. . .	

The third bit of a LPTSL control packet is the *type_id* field, and is always set to a value of 0 for MPEG-2 systems.

The discarding function and its associated list of parameters that is provided by the user on the source machine is sent over to the discarding station. However, sending the function name as a string of characters is not the most efficient way of transmitting over the network. Instead, the LPTSL, at the source station makes an internal call[14] to the function library and obtains the function index corresponding to that function name. Since we do not envision more than 256 different discarding functions, simply using 1 byte suffices as a *function_index*.

The 1 byte *library_version* number is included in the control packet to ensure that the library functions are identical on both the stations. The *library_version* is obtained by executing a *lpfnlibver()* system call.

The parameters that are used by the discarding function are encoded in the control packet as follows:

The function_index is followed by a 1 byte field which gives the number of arguments that follow. Each argument is preceded by a 1-byte length field which gives the number of bytes used to represent it. e.g a short integer parameter has a length field equal to 2, followed by the actual value represented in two bytes.

The format string, *fmtstr* specified in the *lpfn* function call indicates the exact number and the type (and hence their length in bytes) of each parameter.

5.2.6 Modification to the MPEG-2 program_map_table

The MPEG-2 transport stream carries PSI tables which include the *program_map_table* which provides the mappings between program numbers and the description of each elementary stream that is contained within the programs (refer section 5.1.1).

The LPTSL control packet contains information regarding loss profiles that is specific to each elementary bit stream. The information carried in the control packet is, in fact, an enhancement to the description of

[14] refer to function call *fnindex()*.

each elementary stream described in the table. Including the loss profile information in the *program_map table*, enables us to associate each stream with its appropriate loss profile information.

The *program_map_table* is modified as follows :

program_map_table	
table_id	8
.	
.	
program_number	16
.	
.	
program_info_length	12
descriptor	variable
stream_type	8 * N
reserved	3 * N
elementary_PID	13 * N
reserved	4 * N
ES_info_length	12 * N
descriptor	variable * N
LPTSL control packet * N	
CRC_32	32

The multiplying factor denotes the 'N' distinct elementary bit streams that make up the program. Recall that the *program_map_table* describes all the programs that are being carried within the transport stream.

6 Conclusions and Future Work

In a mobile computing environment, whenever a cell gets overloaded with too many user connections, the aggregate bandwidth required exceeds the available bandwidth of the cell's transmitting station. The quality of each connection deteriorates as data is discarded by the cell transmitter to adjust to the reduced available bandwidth. In the absence of any guidelines, the transmitter would discard data in any arbitrary manner. It is highly possible that such a discarding of data results in the loss of some critical portions of the data while retaining portions that are not as important.

A Loss Profile is a user specification that denotes a preferred way in which data can be discarded in an event that bandwidth requirements exceed the available bandwidth. We have defined a new transport sub-layer, the LPTSL which implements loss profiles in various transport systems. In particular, it has been incorporated in the MSP and MPEG-2 transport systems. Simulation studies indicate that the proposed approach yields accurate results and that the LPTSL can be incorporated into existing transport protocols. We are currently implementing LPTSL on top of UDP (and are simultaneously modifying UDP to be more efficient for mobile environments).

References

[1] A. Bakre and B. R. Badrnath,"I-TCP: Indirect TCP for Mobile Hosts", *Technical Report* DCS-TR-314, Rutgers University, Piscataway, NJ 08855.

[2] Paul T. Brady,"A Model for Generating On-Off Speech Patterns in Two-Way Conversation",*The Bell System Technical Journal*, (Sept. 1969), pp. 2445-2472.

[3] D. R. Cheriton, "VMTP: Versatile message transaction protocol" RFC 1045, Stanford Univ, Stanford, CA, Feb. 1988.

[4] R. Ghai and S. Singh,"An Architecture and Communication Protocol for Picocellular Networks",*IEEE Personal Communications Magazine*, Vol. 1(3), 1994, pp. 36-46.

[5] A.C. Hung, "PVRG-MPEG CODEC 1.1", *PVRG, Stanford University*, June 1993

[6] T. Imielinski and B. R. Badrinath,"Mobile Wireless Computing: Challenges in Data Management", *Communications of the ACM*, Vol. 37(10), October 1994, pp. 18-28.

[7] J. Ioanidis, D. Duchamp and G. Q. Maguire, "IP-based protocols for mobile internetworking" *Proc. of ACM SIGCOMM'91*, pp 235-245, September 1991.

[8] T. F. La Porta and M. Schwartz,"The Multistream Protocol: A Highly Flexible High-Speed Transport Protocol", *IEEE Journal on Selected Areas in Communication*, Vol. 11(4), May 1993, pp. 519-530.

[9] T.F. La Porta, "A feature rich transport protocol", Ph.D. dissertation, Columbia Univ. New York, NY, May 1992.

[10] S. Lai, T. Kurita, "Quality Requirements for Multimedia Communication Services and Terminals: Interaction of Speech and Video Delay", *IEEE GLOBECOM '93*, pp. 389-393, 1993.

[11] Craig Partridge, Gigabit Networking, Addison Wesley, 1994.

[12] D.C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems" *IEEE J. Select. Areas Commun.*, May 1993.

[13] K. Seal and S. Singh,"Loss Profiles: a new QOS parameter for mobile computing", Dept. of Computer Science, U. of South Carolina, Columbia, SC 29208, October 1994.

[14] S. Singh,"Quality of Service Guarantees in Mobile Computing", *Journal Computer Communications*, (to appear).

[15] J. Sterbenz and G. Parulkar, "AXON: Application-oriented lightweight transport protocol design" *ICCC '90*, New Delhi, India, Nov. 1990.

[16] F. Teraoka and M. Tokoro, "Host Migration Transparency in IP Networks: The VIP Approach", *SIGCOMM*, Vol. 23, No. 1, Jan 1993, pp. 45-65.

[17] C.A. Thekkath, T. D. Nguyen, E. Moy and E.D. Lazowska, "Implementing Network Protocols at User Level" *IEEE Trans. Networking*, Oct 1993.

[18] J.A. Zebarth, "Let Me Be Me", *IEEE GLOBECOM '93*, Nov.29 - Dec.1, (1993), pp. 389-393.

[19] Grand Alliance HDTV System Specification, Draft document, Submitted to the ACATS Technical Subgroup, Feb. 22, 1994.

[20] Generic Coding of Moving Pictures and Associated Audio, Recommendation H.262, ISO/IEC JTCI/SC29/WG11 N0702, March 25, 1994.

[21] "Transmission control protocol - DARPA Internet protocol program specification" RFC 793, Sept. 1981.

[22] "XTP protocol definition revision 3.4" Protocol Engines Inc., 1989.

[23] ISO/IEC 1-13818 CD, "MPEG-2 Systems Working Draft, Part 1 : Systems", *MPEG Systems Working Committee*, ISO/IEC/JTC1/SC29/WG11 N0601, November 1993

[24] ISO/IEC 2-13818 CD, "MPEG-2 Systems Working Draft, Part 2 : Video", *MPEG Systems Working Committee*, ISO/IEC/JTC1/SC29/WG11 N0601, November 1993

[25] Grand Alliance Specialist Group, "Transport Layer Functional Description", October 1993