A Data Labelling Technique for High-Performance Protocol Processing and Its Consequences

David C. Feldmeier
Computer Communication Research Group, Bellcore
445 South Street, Morristown NJ 07960
dcf@bellcore.com

Abstract

Reordering and reassembly of data before processing can reduce communication system performance as seen by the application. We examine a method of explicitly labelling blocks of data with sufficient information to allow processing of misordered data. Our labelling syntax for data blocks, which we call *chunks*, is cleaner and more general than that of other protocols. We show how chunks can be used for efficient fragmentation/reassembly and compare chunks with other fragmentation systems. End-to-end error detection is complex for chunks or other systems that allow both fragmentation and processing of misordered data. We show that it is possible to design an end-to-end error detection system that does not compromise chunk processing performance. Chunks can take advantage of processing techniques such as Integrated Layer Processing and can be used to implement concepts such as Application Layer Framing [CLAR 90].

1 Introduction

For the last four years, we have been exploring protocols designed for high application-to-application performance. The driving force behind this work has been our participation in the AURORA gigabit network testbed [BIER 92], [CLAR 92]. Often, high performance communication ends at the edge of the network; we are interested in providing high performance communication to the application. By high performance, we mean *both* high throughput and low latency.

We have found that high-performance processing is easiest if a group of data that require identical processing have a completely self-describing header. We call these self-describing groups of data *chunks*. By self-describing, we mean that chunk headers contain enough information that each chunk can be processed by the entire protocol stack without depending on the arrival of any other chunk. We will describe the performance advantages of chunks and describe information that must be contained in chunk headers to allow high application-to-application performance. We will also show that chunks provide an efficient means of performing fragmentation and reassembly. The nature of chunks makes end-to-end error detection complex than for conventional protocols. However, we show that end-to-end error detection is possible while using chunks and describe a specific technique.

The following definitions should be understood before reading the remainder of the paper. A *packet* is the unit of multiplexing in a network and the atomic physical unit of data exchanged between protocol processors. A *protocol data unit* (PDU) is the logical unit of protocol processing exchanged between peer protocol entities. A TCP segment is an example of a PDU. PDU's are be mapped into packets for transmission between protocol processors. As we shall see later, a chunk is a self-describing piece of a PDU and a packet is used as an envelope for carrying chunks.

Chunks are designed to support high-performance protocol processing by allowing packets to be processed at the receiver as they arrive without intermediate buffering for reordering or reassembly. Buffering before processing increases end-to-end latency of data, because of the time that the data are in the buffer. Eliminating buffering also simplifies hardware protocol processors. A major disadvantage of buffering data before processing in RISC workstation architectures is that buffering requires moving the data twice: once from network interface to memory (the buffer) and once from memory to the processor. Because the bus is often a throughput bottleneck on RISC workstations, moving data across the bus twice can decrease protocol processing throughput. The idea of increasing protocol performance on RISC workstations by eliminating buffering in the protocol stack has been called *Integrated Layer Processing* (ILP) [CLAR 90], *lazy message evaluation* [O'MAL 91] and *delayed evaluation* [PEHR 92].

Assuming that the data transmission rate in a system does not exceed the receiver's capacity, processing data as they arrive at the receiver is easy if no misordering occurs. However, data misordering can be caused by message loss in the network. For example, if message 1 is received, message 2 is lost, message 3 is received, and the retransmission of message 2 is received, data have been misordered. The unit of misordering is the PDU of the error control protocol. Another type of misordering is packet misordering in the network caused by multipath routing. For example, obtaining gigabit rates on a SONET OC-3 ATM network requires using eight 155 Mbps ATM connections in parallel. Skew among the routes can cause packets[1] to leave the network in a different order than that in which they entered. Route changes that occur during communication also can cause packet misordering, because the first packet sent along the new route may arrive before the last packet sent along the old route.

Because misordering occurs, many protocols reorder data before processing. Reordering implies that at least some data are buffered before processing, which we would like to avoid. We claim that reordering before processing is not always necessary because some applications can accept misordered data. One such application is bulk data transfer. Regardless of the order in which data arrive, they can be correctly placed in the application address space[2]. Another example is video. Although the video frames themselves must be presented in the correct order, data of an individual frame can be placed in the frame buffer as they arrive without reordering. Another reason why we need not reorder before processing is that there exist protocol operations that provide the equivalent functionality of CRC error detection and DES cipher block chaining encryption, but with the additional property that they can be performed on misordered data [FELD 92].

The only remaining obstacle to the processing of misordered data is to assure that each packet contains sufficient information to determine how to process the packet payload. With current protocols, we run into trouble as soon as some PDU is larger than a packet. The problem is that PDU elements are implicitly identified by their position within the PDU, which means that to process a packet that contains a piece of a PDU requires already having seen all previous pieces of the PDU. A way to avoid this problem is to assure that all PDU's fit into a single packet. However, such an approach is not attractive because placing the control overhead of every PDU in every packet may be inefficient; also, we may not even know the size of the smallest packet in our system if we transmit across an internet. Chunks provide a better solution by explicitly identifying PDU elements, which allows both processing of misordered data and spreading of PDU control overhead across multiple packets.

[1]Packet are *cells*, in this case.

[2]We can think of this as *spatial* reordering versus conventional *temporal* reordering.

2 Data Labelling Format

Chunks are completely self-describing data units, within which all data is processed uniformly. As mentioned previously, conventional protocols identify PDU data and control by their position within the PDU. With chunks, we introduce the notion of explicit data typing within a PDU. In our PDU's, pieces of the PDU are individually labelled with a TYPE field. The explicit TYPE field indicates how the piece of the PDU should be processed. The basic PDU contains pieces with TYPE's of "data" and "control". For example, in a transport-layer PDU (TPDU), the PDU payload would be of TYPE "data" and the error detection code field would be of type "control". PDU's may contain more than one type of control, and thus, multiple control TYPE's may be used.

In addition to a TYPE field, PDU pieces require additional fields for complete identification. For PDU data (TYPE = "data"), a ⟨ID, SN, ST⟩ tuple provides complete identification. The ID identifies the specific PDU to which the data belong, and the SN is the data's sequence number within the PDU payload. The first piece of data of the PDU has a SN of zero, and the last piece of data of a PDU is indicated by an ST bit (STop bit) that is set to one. The PDU payload is also known as a *frame*, and the ⟨ID, SN, ST⟩ tuple is *framing* information.

A single ⟨ID, SN, ST⟩ tuple is sufficient in a system that uses only a single PDU. However, most communication systems break the data stream into different sets of PDU's for different processing functions. For example, a single data stream maybe divided into PDU's one way for error detection purposes, and divided into PDU's another way for encryption purposes. Figure 1 shows how two different processing functions may frame the same data in different ways; a single piece of data in the data stream belongs to both PDU *B* of PDU type 1 and PDU *W* of PDU 2. Thus, we allow multiple ⟨ID, SN, ST⟩ tuples to be associated with a piece of data, one for each different PDU in the communication system.

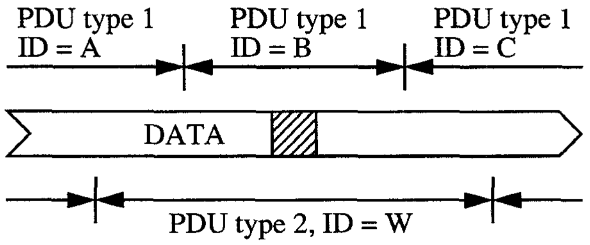


Figure 1: Dividing a data stream into multiple PDU's.

We assume that data streams are uni-directional and that bi-directional streams are constructed with two uni-directional streams. For simplicity, we treat an entire connection as a single, large PDU, and so one ⟨ID, SN, ST⟩ tuple is reserved for the connection. The connection ID is intended to refer to a single, unmultiplexed application-to-application conversation [FELD 90]. The SN and ST function as previously described, with one exception. Unlike other PDU's, SN's of connections are reused over time. The beginning of a connection is indicated with a special signalling messages (connection establishment) rather than an SN of zero.

Although conceptually each piece of data is labelled with a TYPE field and multiple ⟨ID, SN, ST⟩ tuples, a group of data with contiguous sequence numbers that have identical TYPE and ID's can share a single header. Thus, a chunk is a group of data, along with a single header to label the data. The chunk header carries the TYPE and ID's shared by all data of the chunk, the SN's of the first data of the chunk, and the ST bits for the last data of the chunk[3]. Because all chunk data share identical TYPE and ID fields, a single context retrieval is required per chunk and the chunk payload is processed uniformly by all protocol functions. In addition, the chunk header carries SIZE and LEN fields that indicate the size and number of the data pieces in the chunk.

Control information also is carried in chunks. We assume that each type of control information is given a different TYPE and that control information is indivisible, so no LEN field is needed[4]. Control information is associated with only one type of PDU. For example, an error detection code is associated with TPDU's and not encryption PDU's. An example chunk is shown in Figure 2. The data chunks (TYPE = D) contain three PDU types: TPDU, external PDU[5] (X), and connection (C). TPDU's are represented by the ⟨T.ID, T.SN, T.ST⟩ tuple, external PDU's are represented by ⟨X.ID, X.SN, X.ST⟩ tuples, and the connection is represented by the ⟨C.ID, C.SN, C.ST⟩ tuple. Notice that there is some redundancy in the chunk headers; we consider more efficient ways of encoding chunk headers in Appendix A.

Chunks are pieces of PDU's and chunks are moved among protocol processors inside of packets. Packets can be considered envelopes that carry integral numbers of chunks. If a chunk is longer than a packet, it can be split into smaller chunks that fit into packets as shown in Figure 3. Each fragmented chunk has the same TYPE, SIZE and ID (C.ID, T.ID, and X.ID) fields as the original chunk. The LEN and SN fields (C.SN, T.SN, and X.SN) are adjusted appropriately to reflect the contents of the new chunk. Only the chunk that contains the last data of the original chunk has its ST bits (C.ST, T.ST, and X.ST) set to the values of the ST bits in the original chunk; no ST bits are set in any other chunk. The detailed algorithm is in Appendix C. The SIZE field assures that the atomic units of protocol data processing are not split. For example. DES encryption works on 64-bit blocks and we do not want to split these block into two pieces that may arrive separately.

[3]A chunk header contains the ST bits of the last data of the chunk, because shared TYPE and ID's mean that only the last data of the chunk could possibly have any ST bits set.

[4]Control information that is divisible could be carried in chunks similar to data chunks.

[5]External PDU refers to any PDU other than the TPDU. For example, the external PDU could be a PDU of importance to the application (also known as an Application Layer Frame or ALF [CLAR 90]).

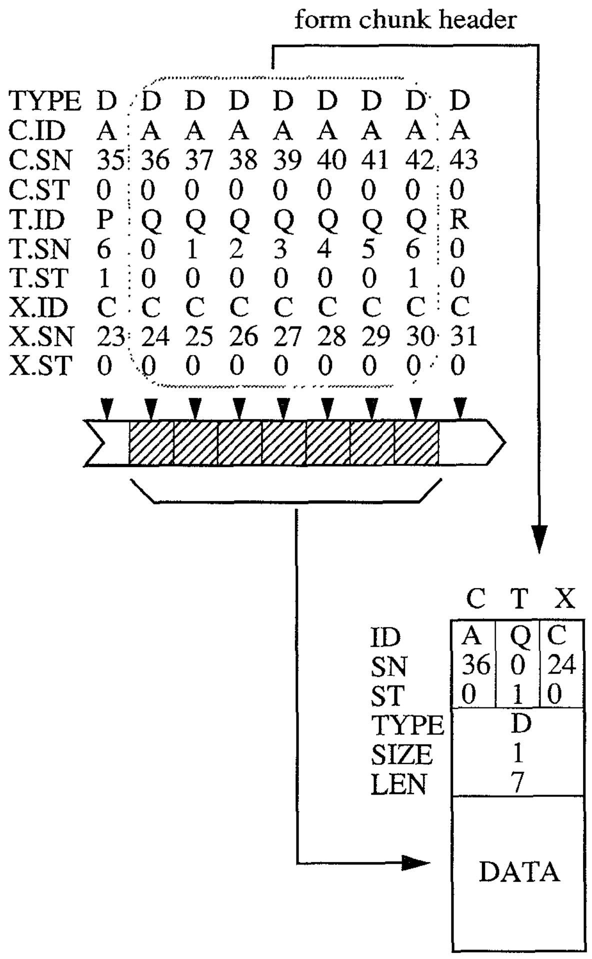


Figure 2: Formation of a TPDU data chunk.

If chunks are smaller than a packet, then as many chunks as fit can be placed in a single packet, as for packet 2 of Figure 3. A TPDU data chunk and a TPDU control chunk (TYPE = ED) that contains the error detection code are combined into a single packet. The chunks are removed from the packet and processed separately at the receiver. Because chunks allow misordering, how the chunks are placed in a packet is irrelevant. Placing multiple chunks in a packet is necessary if we want to send an entire PDU in a single packet. If chunks do not fill a packet completely, we must indicate when the last valid chunk has been reached. A chunk with LEN = 0 is placed after the last valid chunk in the packet.

Chunks provide a syntax only, with no associated semantics. To use chunks to implement a concept such as ALF, we must associate the semantics of an application frame with one of the ⟨ID, SN, ST⟩ tuples in a chunk. Semantics for other types of operations, such as error control, can be associated with other ⟨ID, SN, ST⟩ tuples in a chunk.

The syntax of chunks is similar to the syntax of other protocols, and Appendix B contains a comparison of chunks with other protocols.

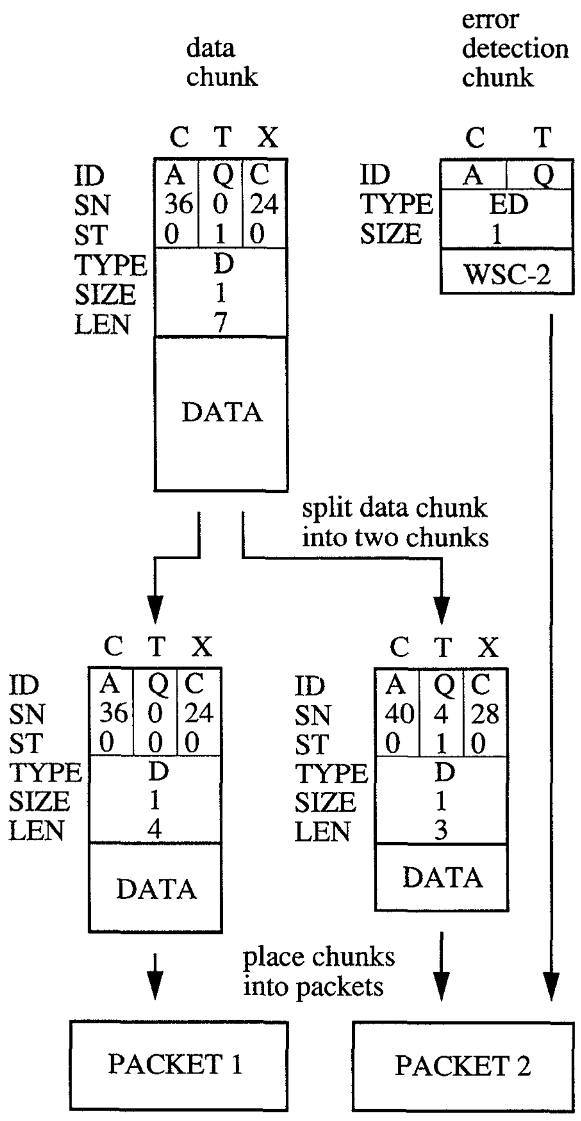


Figure 3: TPDU chunks and their mapping onto packets.

3 Fragmentation

We often want to use PDU's that exceed the packet size of the underlying network. The splitting of PDU's into pieces is called *fragmentation* or *segmentation*. Chunks can be used for a flexible and efficient fragmentation system. In this section, we discuss fragmentation in general and then discuss the advantages of using chunks for fragmentation/reassembly.

Fragmentation is used because it can be more efficient than placing every PDU in a single packet. For example, consider two supercomputers exchanging large blocks of data. For performance reasons, the supercomputers may prefer to do protocol processing on 64 kbyte blocks even though the network packets may be much smaller[6]. Another reason to spread PDU overhead across multiple packets is that the packets may be too small to efficiently carry a PDU per packet; such is the case with ATM cells. Also, interrupts can be reduced if the host-network interface interrupts only after complete PDU's have been received. Such an approach is suggested in [STER 90], and a host-network interface built by Davie moves individual packets across a computer bus using DMA, but generates interrupts only for complete PDU's [DAVI 91].

In an internetworking environment, many different packet sizes may exist. Consequently, we may need to perform fragmentation within the network. Fragmentation can be handled several ways [SHOC 79], [KENT 87][7]:

1. Never fragment - discard packets that are too large
2. Intra-network fragmentation
3. Inter-network fragmentation
4. Never fragment - never send packets larger than a specified maximum size

Discarding packets that are too large (option 1) is unacceptable. Intra-network fragmentation (option 2) can be done transparently, but the approach sacrifices the flexibility of alternate routing [SUNS 77], [SHOC 79]. Also, there can be a problem if different networks use different fragmentation schemes; we may have to reassemble at the exit of one network and re-fragment at the entrance to the next network [SUNS 77]. Inter-network fragmentation (option 3) seems to be the solution of choice for internetworking [CERF 74], [SUNS 77], [SHOC 79].

A specified maximum packet size (option 4) guarantees inefficient use of almost every network except the one network whose packet size is limited to the internetwork maximum size [SUNS 77], [SHOC 79]. Kent and Mogul [KENT 87] argue against fragmentation and for a variation of option 4. They suggested avoiding IP fragmentation by dynamically determining the minimum transmission unit (MTU) for a route. Dynamically determining the MTU for a particular route reduces PDU overhead on routes with MTU's that are larger than the internet-wide MTU. Aside from this one difference, all of their arguments also hold for option 4.

Kent and Mogul claim that fragmentation causes inefficient use of resources, but there is no way to avoid the additional overhead of small packets if we must use a route with small packets. Reassembly within the network can reduce overhead on links of a route after the link with the smallest MTU.

Kent and Mogul argue that fragment loss degrades performance because if a single fragment is lost, then an entire TPDU[8] is retransmitted. However, if such losses occur often enough to be a problem, a good transport protocol implementation should reduce its TPDU size to match the observed network error rate without any direct knowledge of whether

[6]A high-speed TCP implementation on a Cray uses 64 kbyte TCP segments [BORM 89].

[7]Although these references discuss fragmentation in the network, the same ideas apply regardless of where fragmentation is performed.

[8]The TPDU in their case is a TCP segment.

fragmentation is occurring. Also, if fragments travel along the same route, we have the option of dropping all of the fragments of a TPDU if any fragment must be dropped, a technique suggested by Turner [TURN 92].

Kent and Mogul claim that efficient reassembly is hard, and thus network fragmentation should be avoided. However, eliminating network fragmentation does not eliminate reassembly. Even if no fragmentation occurs in the network, we must reassemble (or reorder) packets as they arrive at the receiver to recover the original data stream. Reducing the TPDU size to eliminate network fragmentation does eliminate reassembly of fragments into TPDU's, but at the expense of complicating reassembly of TPDU's because more TPDU's are used. The main advantage of avoiding network fragmentation is that a two-step reassembly process (fragments to TPDU's, then TPDU's to data stream) becomes a one-step process (fragments to data stream).

Kent and Mogul discuss the intentional use of fragmentation, which decreases the amount of fragmentation/reassembly at the data stream level and increases the amount of fragmentation/reassembly at the PDU level. Such an approach makes sense if fragmentation is cheaper at the PDU level (perhaps because different types of processors are available).

3.1 Chunk Fragmentation

Chunks are well suited to fragmentation because chunks preserve all of their properties under fragmentation. Also, regardless of how many different fragmentation steps occur, chunks can be efficiently reassembled in a single step. Chunk fragmentation is easiest to understand if we think of packets as envelopes that carry chunks. Whenever we must change from one packet size to another packet size, it is as if chunks are emptied from one size of envelope and placed in another size of envelope.

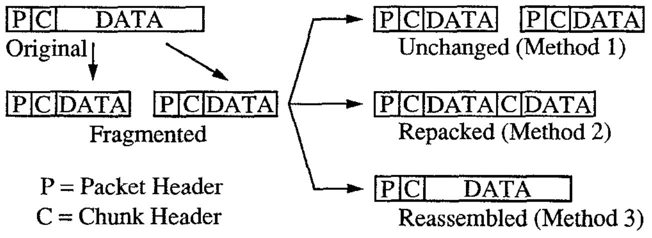


Figure 4: Using chunks for internetworking.

When moving chunks from large packets to small packets, it may be that some chunks are too large to fit into the small packets. If so, the large chunk can be split into smaller chunks that do fit into the small packets, as was shown in Figure 3 and described in Appendix C. Because splitting a chunk forms multiple chunks by definition, the receiver always receives packets filled with chunks, and the format of the received chunks is identical regardless of how much network fragmentation occurs. Splitting chunks can be considered a generalization of existing fragmentation protocols, because multiple $\langle$ID, SN, ST$\rangle$ tuples must be manipulated rather than a single $\langle$ID, SN, ST$\rangle$ tuple. Such manipulation can be done in parallel.

When moving chunks from small packets to large packets, we have the three choices shown in Figure 4:

1. Put one small chunk in each large packet
2. Combine multiple small chunks into a large packet
3. Perform chunk reassembly[9]

With chunks, all three options are possible, and the specific choice is left to the implementor. We can use arbitrary combinations of intra-network and inter-network fragmentation, because chunk fragmentation and reassembly in the network is completely transparent to the receiver. Chunks also allow the possibility of packing unrelated chunks into packets, a technique that is simpler than and almost as efficient as chunk reassembly.

3.2 Chunks versus Other Systems

Most fragmentation schemes take a PDU, split the PDU into pieces and transmit the pieces as PDU fragments. The problem with this scheme is that individual packets (fragments) no longer contain enough information about the framing structure of the PDU to allow misordered packets to be processed as they arrive at the receiver. Thus, fragments must be reassembled into PDU's at the receiver before they can be processed as usual. We would prefer that each fragment could be immediately processed without prior reassembly because reassembly before processing implies buffering.

Unlike chunks, IP fragmentation never combines fragments in the network, and thus, never reassembles or combines fragments inside the network. IP does not perform reassembly, and Cerf and Kahn argue that routers should not perform reassembly for practical reasons [CERF 74]. However, reassembly in routers may be a practical option today. IP is not designed to combine arbitrary fragments in a single packet.

Another advantage of chunks is reduced demultiplexing cost. Because of multipath routing, a mixture of complete PDU's and fragments of PDU's could arrive at the receiver. The receiver must examine the received packet to demultiplex the packets to the appropriate protocol. Although this demultiplexing step is simple, it does have some cost. Chunks are processed identically regardless of whether network fragmentation has occurred.

An alternative to fragmentation is to convert large PDU's into smaller PDU's, as in done in XTP. However, performing this function at just the transport layer is not enough - it must

[9]The algorithm for chunk reassembly is in Appendix D.

be done for all PDU's at all layers. One consequence of this is that all of the higher-layer protocols in use on the network must be at the point of fragmentation, e.g., anyone who fragments XTP packets must understand the XTP protocol. Fragmentation is no longer independent of the upper-layer protocols. Another disadvantage is that the overhead of all PDU's must be carried in each packet, and this could be inefficient.

XTP also has a scheme similar to that of combining multiple chunks in a single packet. An XTP SUPER packet is a packet that contains multiple XTP TPDU's. However, the SUPER packet format is not the same as the regular XTP packet format. Chunks have the same format regardless of what fragmentation, reassembly, or chunk combining may have occurred.

The disadvantage of chunks is that chunk fragmentation requires manipulation of multiple levels of framing information, compared with other fragmentation systems that manipulate a single level of framing information. However, this manipulation is quite simple and can be done in parallel for each level of framing.

Chunks are a compromise between the traditional fragmentation approach and the XTP approach. With fragmentation, entities that convert between two packet sizes do not need to know either the syntax or semantics of higher-layer protocols. The XTP approach requires that both the syntax and semantics of XTP be known to entities that convert from one packet size to another. Chunks provide a syntactic structure that must be understood by any entity that processes chunks. However, intermediate systems that perform fragmentation need not understand the semantics associated with the chunk syntax.

3.3 Reassembly

At some point in the receiver, we must perform complete reassembly of the transmitted data. There are several options:

- Let the application deal with reassembly[10]
- Reorder data before passing to application
- Reassemble data into larger blocks (e.g., complete PDU's) before passing to application

The second and third options are not mutually exclusive. As discussed previously, passing data to the application as it arrives has both latency and throughput advantages over reordering and reassembly. Immediate packet processing minimizes data movement, while reassembly requires two accesses to each piece of data: one access to examine the header for reassembly purposes and one access for protocol processing after reassembly is complete. Reordering is somewhere in-between and the number of times that data must be accessed depends on the amount of misordering in the network. Maximum system performance is achieved if any reordering or reassembly happens as close to the application as possible.

[10] Sometimes called *reassembly in place* [STER 90].

Any of three approaches presented above can be used with chunks because chunks can be reassembled in a single step regardless of whether fragmentation occurs in the network. Thus, chunks can provide the low PDU overhead of fragmentation with the reassembly efficiency of avoiding fragmentation. Chunks have another advantage because they need not be fragmented completely until they reach the router attached to the network with the MTU.

Reassembly buffer lock-up occurs when the reassembly buffer is filled completely and yet no single PDU is complete. Reassembly buffer lock-up can be a problem with misordered IP fragments [KENT 87]. Chunks eliminate this problem because they can be processed and moved to their final destination as they arrive without prior physical reassembly. To reduce degradation caused by fragment loss and fragment timeout problems, retransmitted data should use the same identifiers as the originally transmitted data. An identical technique can be used with chunks.

Regardless of whether we perform physical PDU reassembly, packet reordering, or immediate packet processing, we must perform *virtual reassembly*. By virtual reassembly, we mean keeping track of the received fragments to determine when all of the fragments of a PDU have been received. If physical reassembly is used, virtual reassembly indicates that a PDU is ready for processing. If reordering or immediate packet processing is used, virtual reassembly indicates all pieces of a PDU have been processed incrementally. For example, if we compute an error detection checksum incrementally as fragments arrive, completion of PDU virtual assembly indicates that the incremental checksum is ready to be compared with the received checksum of the PDU.

Virtual reassembly also may be used to reject duplicate data. For example, if we are performing an incremental checksum calculation, we want to avoid processing the same TPDU piece twice, as this may cause the checksum to be incorrect even if no data corruption has occurred. Another reason to reject duplicates is to prevent a corrupted duplicate from overwriting uncorrupted data that has already been received.

Virtual reassembly can be complex if data misordering occurs. Reference [STER 92] suggests the use of VLSI for high-speed virtual reassembly and McAuley describes a VLSI virtual reassembly unit that is based VLSI chip previously built at Bellcore [MCAU 93b].

4 End-to-End Error Detection

In a communication system, we want to use end-to-end error detection, because it provides higher reliability than hop-by-hop error detection [SALT 84]. The transport layer protocol in the host is responsible for end-to-end error detection. Layers above the transport protocol expect that both the data and control parts of their PDU's are protected by the transport protocol.

End-to-end error detection of higher-layer PDU's is conventionally performed by carrying the higher-layer PDU's (framing, control information, and data) as TPDU payload. With chunks, higher-layer PDU framing information is carried in chunk headers, so both chunk headers and payloads must be protected by the error detection code. Because chunk headers are manipulated during network fragmentation, end-to-end error detection requires an error detection code value that is unaffected by the fragmentation procedure. In this section, we describe how to perform end-to-end error detection using chunks. Although we focus on chunks, a similar approach is necessary for any system that allows both processing of misordered data and network fragmentation.

Our end-to-end error detection system example uses a new error detection code, WSC-2, that can be applied to misordered data and has the error detection power of an equivalent *cyclic redundancy code* (CRC)[11] [MCAU 93a]. A WSC-2 encoder takes 32-bit symbols of data and creates two 32-bit parity symbols, P_0 and P_1, such that:

$$P_1 = \sum_{i=0}^{k-1} (i+2) \odot d_i \qquad P_0 = P_1 \oplus \sum_{i=0}^{k-1} d_i$$

where d_i is a 32-bit data symbol and the operations $\sum$, $\oplus$, and $\odot$ are addition and multiplication performed in GF(2^{32}). Acceptable values for i are $0 \leq i < 2^{29} - 2$; if we have less than $2^{29} - 2$ data symbols, the i values left unused are equivalent to encoding a symbol of zero at that i value. Consequently, WSC-2 will work correctly as long as the the error detection protocol specifies which unique value of i should be used for each symbol that is covered by the error detection code. Our error detection system takes advantage of this flexibility.

Now we consider how TPDU errors are detected. We shall detect errors in three different ways: error detection code mismatch caused by virtual reassembly error, error detection code mismatch caused by header or data corruption, and inconsistency among certain TPDU header fields. Table 1 lists the chunk fields of the TPDU chunk shown in Figure 2, whether a field is altered by fragmentation, and how the results of corruption are detected. Table 1 is simply for reference; it need not be read in detail. Recall from before that the TYPE field distinguishes between data and control parts of PDU's, the SIZE field gives the size of the basic data unit contained in a chunk, the LEN field gives the number of basic data units carried by a chunk, the $\langle$C.ID, C.SN, C.ST$\rangle$ tuple is framing information for the connection between two clients, the $\langle$X.ID, X.SN, X.ST$\rangle$ tuple is external framing information (e.g., ALF), and the $\langle$T.ID, T.SN, T.ST$\rangle$ tuple is TPDU framing information.

Field	Changed by Fragmentation?	How Detected?
C.ID	No	Error Detection Code
C.SN	Yes	Consistency Check
C.ST	Yes	Error Detection Code
T.ID	No	Error Detection Code
T.SN	Yes	Reassembly Error
T.ST	Yes	Reassembly Error
X.ID	No	Error Detection Code
X.SN	Yes	Consistency Check
X.ST	Yes	Error Detection Code
TYPE	No	Reassembly Error
LEN	Yes	Reassembly Error
SIZE	No	Reassembly Error
Data	No	Error Detection Code
Control	No	Error Detection Code
ED code	No	-

Table 1: How corruption is detected for various chunk fields.

We now describe in detail the error detection method used for different groups of chunk fields. Corruption of some fields (TYPE, LEN, SIZE, T.SN, and T.ST) will cause virtual reassembly to fail, either because reassembly never completes or because reassembly completes incorrectly such that error detection code mismatch will occur. These fields need not be covered explicitly by the error detection code.

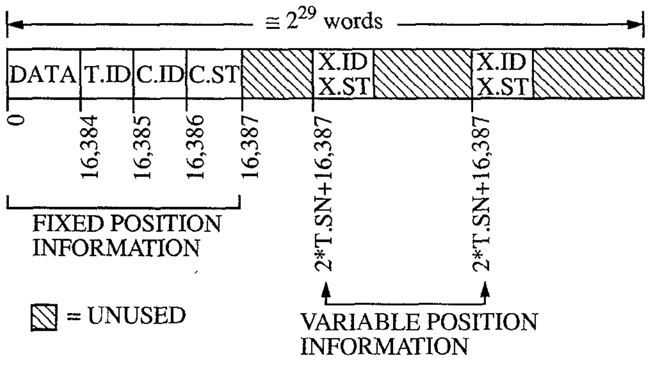


Figure 5: TPDU invariant.

For the fields that are covered by the error detection code, we perform error detection on an invariant of the TPDU under chunk fragmentation. The invariant is simply a way of assuring that the transmitter and receiver perform error detection on the same chunk fields in the same way regardless of network fragmentation. The TPDU invariant is shown in Figure 5[12]. The numbers indicate offset from the beginning

[11]The TCP checksum can be computed on misordered data, but has less powerful error detection properties than both CRC and WSC-2. A CRC cannot be computed on misordered data [FELD 92].

[12]For this example, we show error detection of data chunk payloads only. Any control chunk payloads for higher-layer protocols also must be in the invariant. Control chunk payloads would be treated like data chunk payloads,

of the error detection code space ($i = 0$) in 32-bit symbols. Error detection encoding/decoding is required only for the labelled areas; the shaded areas are unused. We assume that the TPDU data is limited to 16,384 32-bit symbols, and that this information is encoded as symbols 0 through 16,383 of the error detection block. The T.ID and C.ID are constant for all chunks of a TPDU, and are encoded as symbols 16,384 and 16,385 of the error detection block. The C.ST bit can be set only on a TPDU boundary, so a set C.ST bit can occur at most once in a TPDU. The C.ST value is encoded as symbol 16,386 of the error detection block[13].

The X.ID and X.ST fields are more complicated to encode, because multiple X.ID's may exist in a TPDU. For each unique X.ID that occurs in the TPDU, we want exactly one appearance of that X.ID somewhere in the error detection code space. To assure that each X.ID is encoded exactly once, we encode the X.ID field of chunk headers with the X.ST bit set. Because the X.ST bit is set once for each external PDU, the X.ID of that PDU is encoded exactly once. The only remaining X.ID that may not be encoded for a TPDU is the X.ID of an external PDU that begins but does not end within the TPDU. To assure that this X.ID is encoded exactly once, we also encode the X.ID field of chunk headers with the T.ST bit set. Because the T.ST bit is set once per TPDU, the X.ID of the last external PDU of the TPDU is encoded exactly once. Figure 6 shows a TPDU which contains pieces of three external PDU's. The vertical arrows show which PDU boundary triggers the encoding of each X.ID field.

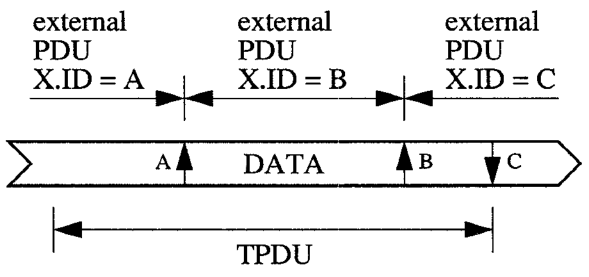


Figure 6: Encoding of the X.ID and X.ST fields.

Because it is possible that the T.ST and X.ST bits are set simultaneously, we must encode the value of the X.ST bit associated with each encoded X.ID field to detect X.ST bit corruption if both the X.ST and T.ST bits are set simultaneously. We assume that the X.ID is 32 bits and the X.ST is one bit, and we encode them within two 32-bit symbols. To assure that no two X.ID/X.ST pairs overlap each other or any other information in the error detection code space, we encode the two symbols starting at position $2 \cdot \text{T.SN} + 16,387$, where T.SN is the T.SN of the data element for which the X.ST or T.ST set.

The two fields we have not discussed are the C.SN and X.SN fields. If the C.SN is uncorrupted, the value of (C.SN – T.SN) is constant for all chunks of a TPDU. If the X.SN is uncorrupted, the value of (C.SN – X.SN) is constant for all chunks of an external PDU within a TPDU. *If not all chunks of a PDU have identical values for either of these differences, then the TPDU is corrupted.*

As might have been expected, the difficult part of end-to-end error detection with chunks is handling the higher-level framing and SN information: the C.SN, X.ID, X.SN, and X.ST fields. Although our example uses chunks designed for one type of external PDU, the same basic idea can be generalized to provide end-to-end error detection of chunks designed for multiple types of external PDU's.

5 Summary

Chunks are self-describing data units designed for high-performance protocol processing. Chunks are typed pieces of PDU's, and packets act as envelopes for carrying chunks across a network. The self-describing nature of chunks allow them to be processed as they arrive at the receiver, regardless of any misordering. The ability to process data without intermediate buffering for reordering or reassembly improves protocol processing performance.

Chunks can be the basis of a flexible and efficient system for fragmentation. Chunks allow fragmentation, reassembly, and the combining of arbitrary chunks into a packet for efficient bandwidth utilization. Chunks can be reassembled efficiently in one step, regardless of how many times they've been fragmented. Convention protocols require a reassembly step for each fragmentation step. Because chunks can be processed in any order, the complexity of managing reassembly buffers is eliminated.

Because chunk headers contain higher-layer framing information, it is not immediately obvious that true end-to-end error detection can be performed on chunks. We describe why end-to-end error detection is difficult for chunks and similar systems, and we show how end-to-end error detection of chunks can be performed using an error detection system that invariant under chunk fragmentation.

Our experience with chunks has shown that they allow protocol implementations with more modularity and parallelism than implementations of protocols with more conventional data structures. A. McAuley of Bellcore has implemented a version of chunks in C, including fragmentation and end-to-end error detection.

The work presented in this paper is just part of the work necessary to support high-performance protocol processing. Much of our previous work has been to provide an infrastructure for supporting chunks, including work describing the advantage of non-multiplexed connections [FELD 90], processing of misordered data [FELD 92], [MCAU 93a] and VLSI reassembly processors [MCAU 93b].

although at a different position in the error detection code space.

[13]The reader may notice that we sometimes use an entire 32-bit symbol to encode chunk fields that may be as small as a single bit; however, with an error detection code space of 2^{29} symbols, sparse utilization is acceptable.

Acknowledgments

My thanks to A. McAuley, with whom I discussed the ideas in this paper while he was implementing a version of chunks. He also read several drafts of this paper. E. Biersack also was involved in discussions about chunks while he was at Bellcore. My thanks also to K. Nahrstedt and one of the anonymous reviewers for their helpful comments.

References

[BIER 92] E. W. Biersack, C. J. Cotton, D. C. Feldmeier, A. J. McAuley and W. D. Sincoskie, "Gigabit Networking Research at Bellcore", *IEEE Network*, 6(3):42–48, March 1992.

[BORM 89] D. A. Borman, "Implementing TCP/IP on a Cray Computer", *ACM Computer Communication Review*, 19(2), April 1989.

[CERF 74] V. G. Cerf and R. E. Kahn, "A Protocol for Packet Network Intercommunication", *IEEE Trans. on Comm.*, COM-22(5):637–648, May 1974.

[CHER 86] D. R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems", *Proc. ACM SIGCOMM '86*, pp. 406–415, Stowe, VT, August 1986.

[CLAR 90] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols", *Proc. ACM SIGCOMM '90*, pp. 200–208, Philadelphia, PA, September 1990.

[CLAR 92] D. D. Clark, B. S. Davie, D. J. Farber, I. S. Gopal, B. K. Kadaba, W. D. Sincoskie, J. M. Smith and D. L. Tennenhouse, "An Overview of the AURORA Gigabit Testbed", *Proc. INFOCOM '92*, Florence, Italy, May 1992.

[DAVI 91] B. S. Davie, "Host Interface Design for Experimental, Very High Speed Networks", *Proc. ACM SIGCOMM '91*, pp. 307–315, Zurich, Switzerland, September 1991.

[DEPR 91] M. de Prycker, *Asynchronous Transfer Mode Solution for Broadband ISDN*, Ellis Horwood, Chichester, England, 1991.

[FELD 90] D. C. Feldmeier, "Multiplexing Issues in Communication System Design", *Proc. ACM SIGCOMM '90*, pp. 209–219, Philadelphia, PA, September 1990.

[FELD 92] D. C. Feldmeier and A. J. McAuley, "Reducing Protocol Ordering Constraints to Improve Performance", B. Pehrson, P. Gunningberg and S. Pink, Eds., *Protocols for High-Speed Networks, III*, pp. 3–17, Stockholm, Sweden, May 1992, North-Holland Publ., Amsterdam, The Netherlands.

[FRAS 89] A. G. Fraser and W. T. Marshall, "Data Transport in a Byte Stream Network", *IEEE Journal on Selected Areas in Communications*, 7(7):1020–1033, September 1989.

[JACO 90] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC 1144, DARPA Network Working Group, February 1990.

[KENT 87] C. A. Kent and J. C. Mogul, "Fragmentation Considered Harmful", *Proc. ACM SIGCOMM '87*, pp. 390–401, Stowe, VT, August 1987.

[LYON 91] T. Lyon, "Simple and Efficient Adaptation Layer (SEAL)", T1.S1.5/91-292, ANSI, August 1991.

[MCAU 93a] A. J. McAuley, "Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes", available via anonymous FTP on thumper.bellcore.com in pub/tp++.

[MCAU 93b] A. J. McAuley, "Parallel Assembly Algorithms for Broadband Networks", available via anonymous FTP on thumper.bellcore.com in pub/tp++.

[O'MAL 91] S. W. O'Malley and L. L. Peterson, "A Highly Layered Architecture for High-Speed Networks", M. Johnson, Ed., *Protocols for High-Speed Networks, II*, pp. 141–156, Palo Alto, CA, November 1990, North-Holland Publ., Amsterdam, The Netherlands.

[PEHR 92] B. Pehrson, P. Gunningberg and S. Pink, "Distributed Multimedia Applications on Gigabit Networks", *IEEE Network Magazine*, 6(1):26–35, January 1992.

[POST 81] J. Postel, "Internet Protocol", RFC 791, DARPA Network Working Group, September 1981.

[SALT 84] J. H. Saltzer, D. P. Reed and D. D. Clark, "End-to-End Arguments in System Design", *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SHOC 79] J. F. Shoch, "Packet Fragmentation in Inter-Network Protocols", *Computer Networks*, 3(1):3–8, February 1979.

[STER 90] J. P. G. Sterbenz and G. M. Parulkar, "Axon: A High Speed Communication Architecture for Distributed Applications", *Proc. IEEE INFOCOM '90*, pp. 415–425, San Francisco, CA, June 1990.

[STER 92] J. P. G. Sterbenz, A. Kantawala, M. M. Buddhikot and G. M. Parulkar, "Hardware Based Error and Flow Control in the Axon Gigabit Host-Network Interface", *Proc. IEEE INFOCOM '92*, pp. 282–293, Florence, Italy, May 1992.

[SUNS 77] C. A. Sunshine, "Interconnection of Computer Networks", *Computer Networks*, 1(3):175–195, January 1977.

[TURN 92] J. S. Turner, "Managing Bandwidth in ATM Networks with Bursty Traffic", *IEEE Network*, 6(5):50–58, September 1992.

[WATS 83] R. W. Watson, "Delta-t Protocol Specification", UCID-19293, Lawrence Livermore Laboratory, April 1983.

[XTP 90] G. Chesson et al., "XTP Protocol Definition, Revision 3.5", PEI-90-120, Protocol Engines Inc., Santa Barabara, CA, September 1990.

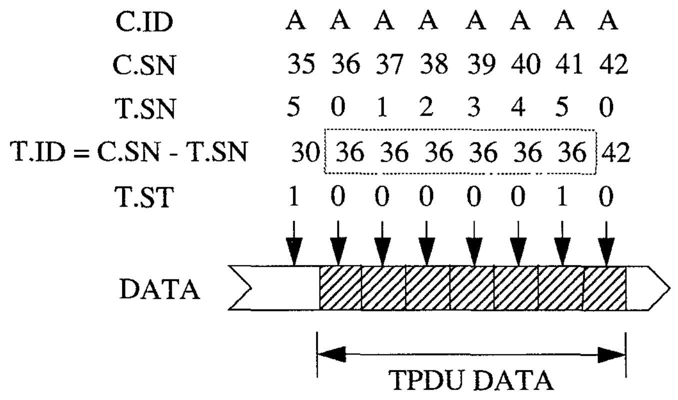
A Implementation Considerations

The simple version of chunks that we have discussed in this paper are easy to parse because of their fixed-field format. Chunks also simplify distributed protocol processing because they can be demultiplexed via the TYPE field and routed to the appropriate processing units. Individual processing units are responsible for knowing which chunk ⟨ID, SN, ST⟩ tuple to use.

Although simple to process, the chunks discussed in this paper do not use bandwidth efficiently. Chunk header size may be reduced if we are willing to perform additional processing and/or take advantage of the characteristics of specific underlying networks. The chunk syntax transformations that we discuss in this section are invertible, because they allow recovery of the original chunk syntax. Protocols can be defined to use the simplest form of chunks and chunk syntax transformations can be used to increase the bandwidth efficiency of chunk headers without changing the basic operation of the protocol. In fact, chunks headers can have different formats in different parts of the network if desired.

One way to reduce chunk header size is to avoid including chunk header fields that seldom change. For example, instead of carrying an explicit SIZE field, chunk size information can be shared by specification or by signalling. With the specification approach, the value of the SIZE field of each chunk TYPE is part of a protocol specification. An alternative is to use a signalling system similar to that used for a virtual circuit; when a connection is formed, the value of the SIZE field of each chunk TYPE can be carried in the signalling message. Another form of signalling system is that used for TCP header compression [JACO 90]; a similar system could be adopted for chunks. With any of these approaches, the chunk header need not contain a SIZE field. The C.ST bit also could be sent as a signalling message, because it is used only when a connection closes.

Some chunk header ID fields can be eliminated with the use of implicit ID's. An implicit ID takes advantage of the fact that the SN fields of a chunk change in lock-step. For example, consider the C.SN (connection SN) and T.SN (TPDU SN) fields of a TPDU. The value of (C.SN – T.SN) is identical for each chunk of a TPDU, and this difference can be used in place of an explicit T.ID (TPDU ID) field. Figure 7 shows an example of how an implicit ID is derived.



C.ID	A	A	A	A	A	A	A	A
C.SN	35	36	37	38	39	40	41	42
T.SN	5	0	1	2	3	4	5	0
T.ID = C.SN - T.SN	30	36	36	36	36	36	36	42
T.ST	1	0	0	0	0	0	1	0

Figure 7: How an implicit T.ID is derived.

Another way to reduce chunk header size is to minimize the number of different frames used by different protocol functions. Because each different frame requires a separate ⟨ID, SN, ST⟩ tuple, the fewer different frames, the smaller the chunk header. Also, each time any frame boundary occurs, a new chunk header is needed. Consequently, aligning frame boundaries also reduces the chunk header overhead.

Packets are utilized more efficiently if multiple chunks can be carried in a packet. Previously we discussed packets that carry multiple chunks from a single connection, and this idea can be extended to packets that carry chunks from multiple connections. Data, signalling information, and acknowledgments can be combined in any combination. Notice that this allows an error detection system that utilizes chunks to achieve the efficiency associated with the piggybacking of acknowledgments without requiring the explicit design of piggybacking into the error control protocol. Thus, chunks provide more modularity in protocol design by allowing some efficiency considerations to be separated from other aspects of protocol design.

In some cases, the chunks headers within a packet may be related; if so, some chunk fields may be implicit rather than explicit. For example, because the chunk following the last TPDU DATA chunk is always a TPDU ED chunk, the ED chunk does not require a chunk header because its TYPE is known, and its C.ID and T.ID fields can be derived from the DATA chunk header. In general, we can use positional information and Huffman encoding to reduce the chunk header overhead within a packet.

On a network that has low loss and maintains packet order, we need not send SN's in each chunk header. SN's can be regenerated at the receiver with a counter that is incremented each time a data element passes. However, because loss and misordering may occur, the counter at the receiver may sometimes lose synchronization with the transmitter. To recover synchronization, the transmitter must send SN information to the receiver occasionally, such as at the beginning of each PDU. During the time that the receiver is out of synchronization, the error detection system will detect the incorrect sequence numbers and allow any incorrect chunks to be discarded.

Chunks are a dynamic way of mapping PDU information to packets. If we make the PDU small enough to fit into a packet, then the same chunks will appear in each packet. If this is true, then a protocol can specify a static packet format rather than the chunk format and eliminate the complexity of dynamically fitting chunks into packets. Thus, the design of conventional packet header formats can be considered a special case of chunks.

B Comparison of Chunks with Other Protocols

The syntax of chunks is similar to the syntax of other protocols, although chunks have a cleaner, more general design. Chunk headers provide explicit framing and type information for all PDU types in a communication system. Existing protocols provide implicit or explicit framing information for one or more PDU types. The equivalent of the chunk SIZE field is implicit for all existing protocols. Carrying some framing information (ID, SN, ST) implicitly is common for protocols designed to operate on non-misordering channels.

The type 5 ATM Adaptation Layer (AAL) [LYON 91] provides a single bit of higher-layer framing information in the ATM cell header that is equivalent to the T.ST bit in chunks. The error detection code field is found by its position in the frame. No explicit ID, SN, or TYPE fields are needed because because ATM links do not misorder. Because no SN is used, an SN of zero cannot be used to indicate the beginning of a frame. A cell is considered to contain the beginning of a frame if the previous cell was the end of a frame. LEN information is explicit.

The type 4 AAL protocol uses an C.ID (MID), a 4-bit C.SN, and framing information denoting the beginning, continuation, or end of message (BOM, COM, EOM) [DEPR 91]. EOM is equivalent to X.ST, and with BOM, the X.ID and X.SN can be derived from the C.SN. No C.ST is used. LEN information is explicit.

The HDLC link-layer protocol[14] provides three levels of framing. The basic HDLC frame is delimited by flags, and the error detection code is found by its position in the frame; thus TYPE, T.ID, T.SN, and T.ST are implicit. HDLC uses a C.ID (address field), C.SN (SN field), and C.ST is indicated by a HDLC disconnect. The P/F bit can be used as an X.ST bit, but the X.ID and X.SN are implicit and derived from the C.SN and X.ST bit. LEN also is implicit.

The URP protocol [FRAS 89] provides three levels of framing. URP uses a C.SN, but the C.ID is implicit because URP connections are mapped one-to-one onto network connections, and the C.ST is indicated by connection tear-down. URP delimits *messages* with a BOT marker (similar to X.ST) and delimits *blocks* (TPDU's) with a BOT marker or BOTM marker (similar to T.ST). The error detection code is found by its position in the frame; thus TYPE, T.ID, and T.SN are implicit. The X.ID and X.SN are implicit and derived from the C.SN and X.ST bit. LEN also is implicit.

Some protocols are designed to work with misordering channels, and provide explicit framing information. IP [POST 81] uses provides T.ID (identification field), T.SN (fragment offset field), and T.ST (logical inverse of the more fragments bit) fields. The VMTP protocol [CHER 86] provides error detection per packet, so T.ID, T.SN, T.ST, and TYPE information is implicit. VMTP also provides an X.ID (transaction identifier), a X.SN (segOffset), and X.ST bit (End-of-Message). LEN is implicit.

Axon [STER 90] provides several levels of framing, Each level of framing has an SN (*index*) and ST bit (*limit*). However, not all levels of framing have an ID, which means that some frames are assumed to be hierarchically nested. Chunks allow the use of completely independent frames at all levels. Axon headers also contain some explicit type information, but some PDU pieces, such as the error detection checksum, have their functionality indicated by position within the PDU, rather than explicit type information. LEN is implicit.

The Axon framing structure provides enough information for placement of misordered data into application memory space. The only data processing that occurs is the computation of an error detection checksum for each packet. The chunks framing structure is designed not only for data placement, but also for data processing functions that are independent of data framing, such as error detection and encryption[15].

Some protocols can accept misordering for some framing levels, but require that data be reordered for other framing levels. The Delta-T protocol [WATS 83] has a C.ID and C.SN, with the C.SN large enough to allow reordering of

[14]Many other link-layer protocols are similar to HDLC (SDLC, ADCCP, LAPB, LAPD, etc.).

[15]The ability to process misordered data depends not only on what framing information is available, but also on the use of functions that can accept misordered data [FELD 92].

misordered data. Within the data stream, Delta-T provides symbols that mark the beginning and end of a higher-level frame (the B and E symbols). The E symbol is equivalent to the X.ST, and the X.ID and X.SN can be derived from the B symbol and C.SN. The XTP protocol [XTP 90] is similar with its use of BTAG and ETAG fields. TYPE, T.ID, T.SN, and T.ST information is implicit for these protocols.

Generally, framing information is provided in two ways: header fields, or flags/symbols in the data stream. The advantage of using header fields is that we need not parse the data stream for flags. The advantage of flags is that multiple frames can be delimited within a single packet. Chunks provide the best of both worlds because multiple chunks, each of which delimits a frame, can be placed in a single packet. Once the chunk headers have been found, we do not have to parse chunk data fields for flags.

C Fragmentation Algorithm

Assume that we have a chunk that we wish to fragment into two chunks, chunk_a and chunk_b. The algorithm below can be repeated until each chunk carries only a single unit of data.

```
If: chunk.len > 1
Then:
    chunk_a.type ← chunk.type
    chunk_a.size ← chunk.size
    chunk_a.len ← new_len
    chunk_a.c.id ← chunk.c.id
    chunk_a.t.id ← chunk.t.id
    chunk_a.x.id ← chunk.x.id
    chunk_a.c.sn ← chunk.c.sn
    chunk_a.t.sn ← chunk.t.sn
    chunk_a.x.sn ← chunk.x.sn
    chunk_a.c.st ← 0
    chunk_a.t.st ← 0
    chunk_a.x.st ← 0
    For: 0 ≤ i < new_len
        chunk_a.data[i] ← chunk.data[i]
    chunk_b.type ← chunk.type
    chunk_b.size ← chunk.size
    chunk_b.len ← chunk.len - new_len
    chunk_b.c.id ← chunk.c.id
    chunk_b.t.id ← chunk.t.id
    chunk_b.x.id ← chunk.x.id
    chunk_b.c.sn ← chunk.c.sn + new_len
    chunk_b.t.sn ← chunk.t.sn + new_len
    chunk_b.x.sn ← chunk.x.sn + new_len
    chunk_b.c.st ← chunk.c.st
    chunk_b.t.st ← chunk.t.st
    chunk_b.x.st ← chunk.x.st
    For: new_len ≤ i < chunk.len
        chunk_b.data[i] ← chunk.data[i - new_len]
```

D Reassembly Algorithm

Assume that we have two chunks, chunk_a and chunk_b, that we wish to reassemble into chunk_c. The algorithm below can be repeated as long as eligible chunks exist.

```
If:
    (chunk_a.type = chunk_b.type) ∧
    (chunk_a.size = chunk_b.size) ∧
    (chunk_a.c.id = chunk_b.c.id) ∧
    (chunk_a.t.id = chunk_b.t.id) ∧
    (chunk_a.x.id = chunk_b.x.id) ∧
    (chunk_a.c.sn + chunk_a.len = chunk_b.c.sn) ∧
    (chunk_a.t.sn + chunk_a.len = chunk_b.t.sn) ∧
    (chunk_a.x.sn + chunk_a.len = chunk_b.x.sn)
Then:
    chunk_c.type ← chunk_a.type
    chunk_c.size ← chunk_a.size
    chunk_c.len ← chunk_a.len + chunk_b.len
    chunk_c.c.id ← chunk_a.c.id
    chunk_c.t.id ← chunk_a.t.id
    chunk_c.x.id ← chunk_a.x.id
    chunk_c.c.sn ← chunk_a.c.sn
    chunk_c.t.sn ← chunk_a.t.sn
    chunk_c.x.sn ← chunk_a.x.sn
    chunk_c.c.st ← chunk_b.c.st
    chunk_c.t.st ← chunk_b.t.st
    chunk_c.x.st ← chunk_b.x.st
    For: 0 ≤ i < chunk_a.len
        chunk_c.data[i] ← chunk_a.data[i]
    For: chunk_a.len ≤ i < chunk_b.len
        chunk_c.data[i] ← chunk_b.data[i - chunk_a.len]
```