Chapter 6

ALGORITHMS FOR PO/PR PROTOCOLS

6.1 Introduction

In this chapter, we present the design and analysis of algorithms and data structures for the implementation of a Partially-Ordered/Partially-Reliable (PO/PR) Transport Protocol. Specifically, we present algorithms for:

- verifying that the sending application submits TSDUs in a valid sending order (a linear extension of the partial order specified by the application),
- resequencing out-of-order packets at the PO/PR receiver, and
- declaring objects lost according to the semantics of the PR reliability class defined in Section 2.8.2.

We also discuss how these algorithms can be extended to support the stream abstraction[65] and explicit release[66]. Finally, we present the linear extension algorithm that is currently implemented in the ReMDoR parser, and discuss whether this algorithm might be better placed in the PO/PR transport sender.

---

[65] See Section 2.3 for a description of the stream abstraction.

[66] See Section 2.6 for a description of explicit release.

6.1.1 Organization of this chapter

This chapter is organized as follows. Section 6.1 provides a context for our discussion by defining the problems listed above more formally, and comparing these problems with the corresponding ones in the standard ordered/reliable transport protocol TCP. We briefly outline how the corresponding problems are addressed in the de-facto reference implementation of TCP, namely BSD Unix[67]. Section 6.2 describes a basic algorithm for topologically sorting a directed acyclic graph (DAG-TS); this basic algorithm forms the basis of most of the other algorithms described in this chapter.

Sections 6.3-6.5, in a sense, "tell the story" from beginning to end of how a ReMDoR document makes its way through the PO/PR transport protocol:

- Section 6.3 describes the modified (DAG-TS) algorithm used by the ReMDoR parser to compute a linear extension of the partial order for transmission of the document. The algorithm is modified in two ways: (1) it must take into account the stream abstraction, and (2) it must take into account the fact that transmission of audio packets requires a certain minimum bandwidth guarantee.

- Section 6.4 describes the modified (DAG-TS) algorithm used by the PO/PR sender to verify that the sending order used by the application is a valid linear extension of the partial order. We also explain why verification must be done to prevent protocol deadlocks.

- Section 6.5 describes the algorithm used by the transport receiver to resequence arriving PDUs for delivery; that is, an algorithm to topologically sort arriving PDUs according to the partial order. We describe both the basic algorithm, and how it is modified to support explicit release. We also compare this approach with a matrix-based

---

[67] As our source for the reference implementation for TCP/IP, we use the annotated source code for 4.4BSD-Lite as it appears in (Wright and Stevens, 1995), since it is widely available and, due to Wright and Stevens' efforts, excruciatingly well documented.

algorithm, and with the algorithm used for resequencing in the reference implementation of TCP.

Sections 6.6–6.8 describe how the DAGITS algorithm can be modified to support the PR reliability class[68] of POCv2.

- Section 6.6 describes the challenges involved in implementing the PR reliability class, and integrating it with the explicit release and stream abstraction features.
- Section 6.7 describes how to adapt the DAGITS algorithm to support PR without explicit release, without changing its worst-case running time.
- Section 6.8 describes an algorithm to support PR with explicit release, with a worst-case running time that is slightly worse, but still acceptable (i.e., it is no worse than that of TCP.)

Section 6.9 surveys the advantages and disadvantages of various ways a partial order can be represented for processing and for transmission across the network. Section 6.10 provides a chapter summary.

6.1.2 Processing overheads in O/R and PO/PR transport protocols

Processing overheads in transport layer protocols have been studied previously in (Clark et al., 1989). The main conclusion of this work is to recommend a two-path design: a highly optimized fast path for packet processing that optimizes the normal case, with a slower path that takes care of the unusual cases.

The underlying concern of all work in this area is to keep the per-packet processing small, so that the transport layer keeps up with arriving PDUs. Therefore, any lengthy processing associated with a single PDU is to be avoided. Avoiding lengthy processing times associated with a single packet operation is also crucial when

---

[68] See Section 2.8.2

a user-space implementation with cooperative multitasking is used, as is the case for UTL (see Chapter 3.)

The author raised this concern in the context of early specifications of POC, which represented the partial order as the transitively-closed adjacency matrix. While this is an efficient representation in terms of bits (upper triangular form can be used by renumbering elements such that 0,1,2,…,n-1 is a linear extension), it requires an $O(n)$ computation for each `Read()` or `Write()` operation to perform operations such as checking the deliverability status of messages. As part of this research, therefore, the author investigated more efficient algorithms for performing the computations necessary for PO/PR service.

6.1.3 Packet resequencing at the transport receiver in ordered protocols

One of the most time consuming parts of the processing for an ordered protocol is the resequencing of out-of-order packets. Essentially, the problem is one of implementing a priority queue. The standard data structure for this purpose, a min heap, provides $O(\log k)$ running time, where k is the current number of outstanding packets. However, due to some practical considerations, resequencing is seldom implemented using a min-heap. First, when a packet arrives, it is necessary to determine whether or not that packet is a duplicate, which involves searching the buffer. Since a heap does not provide an efficient search mechanism, a heap alone is an inappropriate choice.

One might therefore consider a binary search tree. However, it turns out that in practice, an ordered linked list is the most appropriate data structure. There are two reasons why this is so. First, an essential principle of efficient transport protocol design is to optimize the common case. The common case for most transport protocol implementations is that the next packet to arrive is the one expected. If the buffer of out-of-order packets is maintained as a simple sorted linked list, adding the expected packet to that buffer is an $O(1)$ operation; indeed it involves only a handful of instructions.

Second, the length of this list can be bounded by a constant: namely, the window size. In the initial design of TCP, the maximum window size was 64K, and the common packet size was 512 bytes. With these parameters, at most, 128 packets could be outstanding for any given connection. Until quite recently, a far more common window size was between 4-24K, that is, between 8 and 48 packets. An $O(k)$ search as the "rare" case is not very objectionable when k is this small, and thus the overhead and complexity of a binary search tree is not attractive. Indeed, a careful reading of the annotated TCP source code in (Wright and Stevens, 1995) shows that a linear search is used.

We conclude from the preceding arguments that any packet processing that is of similar magnitude (that is, $O(k)$, where k is the number of outstanding packets) should be considered good enough for practical purposes.

6.1.4 The goal for resequencing in PO protocols: $O(1)$ per operation

In terms of packet resequencing in the transport receiver for partially ordered service, the Holy Grail would be to make the processing of each packet $O(1)$. However, given that we need to ensure that every constraint in the partial order is satisfied, the lower bound for n packet operations is $O(e/n)$ per operation (amortized). This precludes the possibility of finding an $O(1)$ algorithm for the general class of all partial orders.

Note however, that if we restrict the class of partial orders, we can find a tighter worst-case bound. Note, for example, that for a chain partial order, $O(e/n)$ reduces to $O(1)$. We now define a more general class of partial orders for which we find an $O(1)$ algorithm:

> **Definition 6.1.1**: A partial order has ***bounded-out-degree d*** if and only if every vertex in the corresponding transitively reduced precedence graph has out-degree $\leq d$.

We will show that for bounded-out-degree partial orders, in addition to bounding the overall running time as $O(n+e)$ amortized, we can also bound the worst-case running time of each single protocol operation by $O(d)$, *not* amortized. For ReMDoR, d corresponds to the number of successors of any element in a PMSL document.

One might ask whether it is useful to distinguish between $O(n)$ and $O(d)$, since in the worst case, $O(d) = O(n)$. Consider, for example, a partial order consisting of an antichain of $\lceil n/2 \rceil$ elements, all of which precede all of the remaining $\lfloor n/2 \rfloor$ elements. However, we conjecture that for multimedia documents, most human authors would limit the number of successors for any given element to a reasonably small value; usually no more than 10, and rarely more than 30, which may in practice be much smaller than n. Thus, for multimedia document retrieval, it is useful to distinguish between a worst case of $O(d)$ vs. a worst case of $O(n)$.

More significantly, the distinction between $O(n)$ and $O(d)$ provides a yardstick for measuring whether the PO/PR algorithms developed in this work are practical for applications other than multimedia document retrieval. For instance, suppose that it is suggested that PO/PR protocols are useful for some application *X*. We can then compare the expected sizes of n and d in the partial orders that arise in application *X*, with those that arise in multimedia documents. used in the experiments of Chapter 5, or any future experiments that are carried out using ReMDoR. Comparing these sizes allows us to develop hypotheses about whether the algorithms developed in this work will be practical in the context of application *X*.

6.2 Topological sort of a directed acyclic graph by incremental delete (DAGITS)

Nearly all of the algorithms described in this chapter are variants of a specific algorithm for the problem of topologically sorting of a directed acyclic graph (TS-DAG). We call this basic algorithm DAGITS: (directed acyclic graph incremental topological sort). The main feature of DAGITS that distinguishes it from other approaches to topological sort is that the linear ordering is produced incrementally, and we can bound the worst-case running time of each incremental step. We assume that the underlying algorithm of DAGITS is well known, since (Cormen et al., 1990) assigns the description and analysis of this algorithm as an exercise[69]. Our contribution is therefore not the DAGITS algorithm itself, but rather

- the characterization of this algorithm as an *incremental* process, along with
- the analysis of the *individual steps,* and
- the application of our incremental characterization of this algorithm to the efficient solution of problems arising in the provision of PO/PR transport service.
    - The remainder of Section 6.2 proceeds as follows.
- Section 6.2.1 outlines the basic TS-DAG problem, and a variant that determines if the input graph is acyclic or not (TS-DAG-V).

---

[69] Exercise 23-4.5, p. 488. The name “DAGITS”, however, is our suggestion.

- Section 6.2.2 describes how the TS-DAG and TS-DAG-V relate to PO/PR transport protocols processing.

- Section 6.2.3 motivates an incremental version of the TS-DAG problem (INCR-TS-DAG) based on the requirements of PO/PR transport protocols.

- Section 6.2.4 shows how a simple and efficient algorithm for TS-DAG (reverse finishing time of a depth-first search, one of the most commonly presented approaches to this problem in introductory algorithms texts) is not suitable as a basis for INCR-TS-DAG.

- Section 6.2.5 presents pseudocode for the DAGITS algorithm for TS-DAG and TS-DAG-V as suggested by (Cormen et al., 1990). In this algorithm, vertices with in-degree zero and their outgoing edges are successively added to the linear ordering, and incrementally deleted from *G*.

- Section 6.2.6 provides a worst-case running time analysis of the DAGITS algorithm for TS-DAG and TS-DAG-V.

- Section 6.2.7 shows how to adapt the DAGITS algorithm to the INCR-TS-DAG problem, and then presents the most important result of this section:

  > *We show that for DAGITS, the worst-case running time for incrementally computing each vertex in the linear ordering is bounded by the out-degree of the vertex returned.*

As explained in Sections 6.2.4 through 6.2.7 below, it is this property the DAGITS algorithm—the bound on each step in the incremental computation—that makes it suitable as the basis for processing algorithms in PO/PR transport protocols.

6.2.1 The TS-DAG and TS-DAG-V problems

Problem TS-DAG: Topological sort of a directed acyclic graph

Input: a directed acyclic graph $G=(V,E)$

Output: a linear ordering of the vertex set V such that if edge $(u,v) \in E$ then u appears before v in the ordering (Cormen et al., 1990.)

We often will relax the assumption that the input is acyclic, and include determination of this in the problem itself:

Problem TS-DAG-V: TS-DAG with validation of input

Input: a directed graph $G=(V,E)$

Output: if G is acyclic, a linear ordering of V as in TS-DAG; otherwise, an error indicating that the graph contains a cycle.

Equivalently, we can characterize any algorithm for topologically sorting a DAG as an algorithm to produce a linear extension of the partial order induced by the DAG (see Section 2.2).

6.2.2 Topological sorting and PO/PR transport protocols

To motivate our discussion of topological sorting, this section briefly describes two problems related to topological sorting which arise in the implementation of PO/PR transport protocols: (1) sending order validation, and (2) resequencing TPDUs for delivery. We discuss these and several other problems in more detail in Sections 6.3 through 6.6.

Sending order validation in PO transport service (PO-SND-VALID)

For partially-ordered transport service (regardless of reliability), the transport sender must determine whether the sequence of objects submitted via

`Write()` operations constitutes a legal sending order—that is, a valid linear extension of the partial order. This problem is equivalent to determining whether a given permutation of the elements of a vertex set *V* is a valid topological sort of a DAG *G*=(*V*,*E*).

The PO/R receiver TSDU resequencing problem (PO/R-RCV-RESEQ)

For partially ordered/reliable (PO/R) transport service, the transport receiver must be able to efficiently:

(a) resequence arriving TPDUs for delivery as TSDUs according to the partial order,

(b) suspend delivery when the partial order constraints would prevent the delivery of any of the TSDUs that have already been received, and

(c) efficiently detect whether the arrival of a particular TSDU permits delivery to resume.

This processing is essentially a form of incremental topological sorting. However, unlike the general TS-DAG problem where the algorithm may freely choose from among multiple linear orderings of the input graph if such exist, PO/R-RCV-RESEQ imposes an extra constraint. Suppose that at some step in an ordinary TS-DAG algorithm, a particular vertex *u* is chosen as the next one to add to the linear ordering. In the PO/R-RCV-RESEQ problem, vertex *u* may be as yet unavailable—that is, it may correspond to a TSDU for which the TPDU has not yet been communicated to the transport receiver. The PO/R-RCV-RESEQ problem requires that in this case, if *any* other vertex *v* is a legal next vertex, it must be selected immediately rather than suspending the topological sort until vertex *u* becomes available. Thus, the linear extension that emerges from an algorithm to solve the PO/R-RCV-RESEQ is not an arbitrary choice from among all the linear extensions of that PO; rather, the selection is constrained by the arrival times of the TPDUs.

6.2.2 Incremental TS-DAG (INCR-TS-DAG)

As stated previously, the transport receiver must keep up with arriving PDUs, and avoid creating application bottlenecks. Therefore, it is crucial to establish an upper bound for the worst-case running time of each atomic operation in a transport protocol implementation. The running time of the TS-DAG algorithm can be divided into atomic operations as follows:

Problem INCR-TS-DAG: Incremental topological sort of a DAG

Operations:

init(*G*)initialize data structures representing a directed acyclic graph *G*=(*V*,*E*) supplied by the caller.

v = *next*() returns the next element in a linear ordering of *V*, or **nil** when the entire set of vertices has been exhausted.

The TS-DAG program is thus broken down into a sequence of (n+1) operations starting with a call *init*(*G*) followed by *n* calls to *next*(). This INCR-TS-DAG version of the problem allows us to pose two questions about any algorithm proposed to solve it:

- what is the worst-case running time of the entire sequence of operations?
- what is the worst-case running time for a *particular* operation in the sequence?

The second question is the crucial one for PO/PR transport protocol processing. To provide for efficient scheduling between the application and transport layer processes, we must bound the *worst-case* running time for individual *init*(G) and *next*() operations.

6.2.3 The usual DFS-approach to TS-DAG is problematic for INCR-TS-DAG

A widely-used algorithm for TS-DAG appearing in many algorithms textbooks computes a linear ordering in time $O(n+e)$ (where $n=|V|$, $e=|E|$) based on the reverse finishing times of a depth-first search of G (Cormen et al., 1990). While this algorithm is simple and efficient, it has a property that makes it unattractive for our purposes: the entire set of vertices and edges must be processed before the first element in the ordering can be produced. If we nonetheless do solve INCR-TS-DAG using the reverse finishing time approach, the overall running time for a sequence of (n+1) operations starting with a call *init*(G) followed by n calls to *next*() will be $O(n+e)$. However, in this solution, we cannot bound, in the worst case, the cost of any *particular* call to the *next*() operation. If a straightforward implementation of the reverse finishing time DFS algorithm is used, either the first call to *init*(G) or the first call to *next*() will require the entire $O(n+e)$ running time, while the successive calls require only $O(1)$ time. The $O(n+e)$ running time is required because it is necessary to run the algorithm to determine the first element in the sorted list from the input as given, storing all the sorted vertices on a stack as they are finished. Successive calls to *next*() are then implemented as *pop*() operations. We can try to improve the average case by using an approach where we apply the algorithm to G^T, reversing edges only as we encounter them rather than in a pre-processing step. The startup cost is $O(n+e)$, just as before. However, the worst-case running time of the initial operation would be unaffected: consider, for example, the operation of this algorithm on the DAG corresponding to a linear order.

6.2.4 Topological sorting by incremental delete and the DAGITS algorithm

The basis for what we refer to as the DAGITS algorithm is the repeated delete approach to topological sorting outlined in (Cormen et al., 1990), exercise 23.4-5, as shown in Figure 6.1.

```
while (there exist  vertices in the graph)
{
    find a vertex v with in-degree 0;
    output v;
    remove v and all of v's outgoing edges from the graph;
}
```

Figure 6.1 Topological sort via repeated delete

(Cormen at al., 1990) poses the problem of implementing the algorithm in Figure 6.1 in $O(n+e)$ time, and asks what happens if the input contains cycles. The DAGITS algorithm is an application of the algorithm in Figure 6.1 to the INCR-TS-DAG problem posed earlier. The DAGITS algorithm requires the data structures shown in Figure 6.2

DAGITS Algorithm: Data Structures:

int:	*in-degree*[*i*]	// current in-degree of vertex [*i*]
list of vertices:	*adj-list*[*i*]	// list of outgoing edges from vertex [*i*]
queue of vertices:	*front*	// queue of vertices with in-degree 0.

Figure 6.2: DAGITS (DAG incremental topological sort)

We ignore the cost of initialization because our purpose is to highlight the time required by each successive invocation of the *next*() operation, which is implemented as shown in Figure 6.3.

```
DAGITS Algorithm:
operation next():
    if (front.isEmpty())
      return nil; // indicates that sort has finished successfully
    vertex u = front.dequeue();
    foreach v in adj-list[u]   // (u,v) is an edge in G
    {
            if (in-degree[v] == 0)
               halt-with-error; // input was inconsistent, or graph contains a cycle
            decrement in-degree[v];  // logically, removes (u,v) from G;
            // optionally, we can also remove v from adj-list[x], but this is unnecessary

            if in-degree[v] ==  0 // v can now be added to the sorted list
                front.enqueue(v);
    }
    return u;
```

Figure 6.3: **operation** *next()*

6.2.5 The DAGITS algorithm: proof of correctness and running time

Theorem 6.2.1: The values returned by successive calls to the *next*() operation constitute a topological sort of graph *G*.

Proof: The items that are initially in the *front* list will be the first values returned by *next*() operations. Since these items have no predecessors in the DAG, any permutation of these values is by definition a legal prefix for the topological sort. It now suffices to prove the following three lemmas, which we do below.

Lemma 6.2.2 The order in which the remaining vertices are enqueued onto the *front* queue is an order consistent with a topological sort.

Lemma 6.2.3 Every vertex will be added to the list; that is, we cannot return **nil** without having first output every vertex in the graph.

Lemma 6.2.4 The value **nil** will be returned on the (*n*+1)th call to *next*().

Proof of Lemma 6.2.2 At the step in the algorithm where each vertex is enqueued, its logical in-degree at that step is zero. This implies that every incoming edge to this vertex has been logically removed from the graph. Since an edge (*u*,*v*) is removed only at the step where *u* is returned as the result of a *next*() operation, this implies that before any *v* can reach the head of the *front* queue, all vertices *u*, such that there exists an edge (*u*,*v*), have already been returned as the result of a *next*() operation (which matches the definition of a topological sort) Therefore, the order of values returned is a legal topological sort of graph *G*. ❑

Proof of Lemma 6.2.3 (By contradiction.) Suppose there were some vertex *x* in a connected component that had not been output by this algorithm before **nil** was returned. Since all vertices with zero predecessors are added to the *front* queue initially, it must be the case that *x* has at least one predecessor. Furthermore, if *x* is never output, this implies that at least one of *x*'s predecessors is never output, since if all of *x*'s predecessors were output, *in-degree*[*x*] would be decremented to zero, and *x* would be output. We can now make the same argument concerning the predecessor of *x*. However, this cannot repeat indefinitely: since G is acyclic and the number of possible edges is finite, this implies that there must be some ancestor of *x* that has no predecessor, but was never output. Since this contradicts the fact that all vertices with zero predecessors are output, we conclude that every vertex is output before **nil** is returned. ❑

Proof of Lemma 6.2.4 Since every vertex is returned at least once, to show that **nil** is returned after exactly *n* vertices are returned, it suffices to show that each vertex can be added to the *front* queue at most once. A vertex *w* is only a candidate to be placed in the *front* queue when the algorithm is processing an incoming edge incident on that vertex. Given that we have already established in (1) above that the order of vertices returned is a legal topological sort at the step where the vertex *w* is added, it is impossible for the algorithm to process any more edges (*v*,*w*). If (*v*,*w*) were to be processed at this point, it would imply that vertex *v* was about to be added to the *front* queue following *w*, which would violate the topological sort property. Therefore, once a vertex has been added to the *front* queue, it is not possible for it to be added a second time. ❑

Theorem 6.2.2: The worst-case running time for a sequence of *n* calls to the *next*() operation for a given graph *G* is $O(n+e)$.

Proof: There are *n* calls to the *next*() operation; which gives us the $O(n)$ term. All operations within each call to *next*() take $O(1)$ time except for the loop on the adjacency list of each vertex as it is encountered. Since every vertex in the graph is processed exactly once, every edge is processed by this loop exactly once, which explains the $O(e)$ term. ❑

Theorem 6.2.3 The running time of the *next*() operation is:

- $O(d)$ for the first n *next*() operations returning an element, where d is the *out-degree*() of the vertex returned by the *next*() operation.
- $O(1)$ for the final *next*() operation.

Proof: The proof is by trivial inspection of the algorithm: all operations take $O(1)$ time except for the loop over the adjacency list of each vertex. Since every step in this loop takes constant time, the entire loop takes time proportional to the out-degree d of the vertex returned. ❑

6.2.6 Concluding Remarks

In Section 6.2 we introduced a particular way of doing topological sort and called it the DAGITS algorithm. The key property of this algorithm is the *incremental* nature of the algorithm; we can divide the total processing time of $O(n+e)$ into discrete steps, each of which takes time $O(d)$ where d is the out-degree of the vertex returned at each step. In the remainder of the chapter, we will see several adaptations of this algorithm to problems that arise in the implementation and use of PO/PR transport protocols.

6.3 Choosing a linear extension in ReMDoR

The first problem we consider is the application layer processing that must be performed before a PO/PR transport service can be used. The POCv2 implementation (as represented in this case, by the prototype in the POL layer of UTL) expects the partial order part of the service profile to be provided in transitively reduced form. Since the techniques to transitively reduce a directed graph can be found in any algorithms text (e.g.,Cormen et al., 1990), we will not address this issue further.

Of greater interest is the problem of selecting a linear extension. If all linear extensions of the partial order are equally acceptable in terms of performance, then finding a suitable linear extension constitutes nothing more than performing a a topological sort over the elements of the partial order, and outputting the sorted list. However, there are several complicating factors.

First, the topological sort algorithm must be adapted to handle the stream abstraction (see Chapter 2) For the most part, this is an inconsequential change. More difficult than extending the algorithm to handle the stream abstraction is dealing with the fact that the choice of linear extension cannot be made in an arbitrary fashion, as it can in the general case of topological sorting. In the usual formulation of the topological sort problem, the algorithm may choose freely from among equally valid topological sorts, or, in partial order terms, linear extensions. (From here on, we will abuse the notation somewhat by freely interchanging the terms *topological sort* and *linear extension*). However, for several reasons, we need a topological sorting algorithm with a bit more intelligence:

(1) (Marasli et al., 1996b) showed that strategic selection from among several linear extensions may provide improvements in performance.

(2) According to the semantics of partial order delivery, all linear extensions may be *acceptable;* however, from an end-user perspective, not all are equally *desirable*. For example, in the `img8par.pmsl` document used in Chapter 5, it would be legal to send the entire first image, then all of the second image, etc. However, the partial order indicates that the author intends the eight images to be interleaved rather than sent sequentially in what would constitute a linear order.

(3) Most importantly, when audio is included, it is essential to ensure that the linear extension provides a sufficient fraction of the available bandwidth to the audio stream so that underflows

> do not occur. Specifically, as data are interleaved, the overall fraction of the bandwidth for audio must not drop below 64Kbps for any sustained length of time.

A full exploration of linear extension selection would form the basis of another complete dissertation. In its full generality, optimal linear extension selection for pre-fetching multimedia documents over constrained bandwidth lengths becomes an NP-complete scheduling problem as illustrated by work on the DEMON project at Bellcore (Rosenberg et al., 1992a, 1992b; New et al., 1992). However, for purposes of evaluating the performance of partially-ordered transport service, a simple heuristic suffices. Therefore, in this work, we address only items (2) and (3) above, and defer item (1) to future work.

The ReMDoR parser uses a variation of the DAGITS algorithm called ReMDoR-LESTAB (Linear Extension Selection w/Target Audio Bandwidth), shown in Figures 6.4 through 6.6) to produce a linear extension of the underlying partial order over individual TSDUs (cells) which is implied by the partial order at the stream object level. This linear extension also guarantees a minimum bandwidth allocation for audio designed to avoid underflow. As shown in Figure 6.6, the *finish*(*u*) procedure decrements the successors of *u*, in the same manner as in the DAGITS algorithm. If any successor of *u* ends up with an in-degree of zero as a result, the *finish*(*u*) procedure also either adds that successor to the *front* queue, or makes it the *currentAudioElement*, as appropriate.

Regarding the parameter *targetAudioBandwidth,* note that the nominal value for this parameter should be 64kbps, which is the bandwidth requirement for the 8Khz μ-law audio encoding used by ReMDoR. In practice, we use a higher value to provide a margin of safety, since the actual bandwidth provided will oscillate around this target value. Table 6.1 shows the parameters used for the performance experiments involving audio cited in this dissertation (see Chapter 5). Note that the actual targetAudioBandwidth values used are larger than 64kbps. The practical effect of a particular value is to reserve either a particular fraction of the true available bandwidth, for example, 62.5% in **R4**, and 50% in **R5**. Future work might incorporate a more sophisticated approach to this scheduling; however, this simple heuristic suffices for our purposes.

Summary of Section 6.3

In this section we have described how the DAGITS algorithm can be adapted to select an appropriate linear extension for transmission of PDUs via ReMDoR, incorporating the stream abstraction, and the need to provide a minimum bandwidth for audio. Future work in this area may incorporate the linear extension optimization techniques described by (Marasli et al., 1996b) to improve performance.

Table 6.1 Audio Parameters Used for Chapter 5 performance experiments

Experiments	*targetAudioBandwidth*	*totalBandwidth*	fraction reserved for audio
R4.1, **R4.2**, **R4.3**, **R4.4**	80kbps	128kbps	80/128=0.625
R5.1 , **R5.2**	128kbps	256kbps	128/256=0.5

Algorithm ReMDoR-LESTAB:

Inputs:

- a DAG *G*=(*V*,*E*), represented as for the DAGITS algorithm (Figures Algorithm 6.2.2, Algorithm 6.x), where each vertex is associated with:
  - *type*: a data type (audio or non-audio)
  - *APDUlist*: a list of APDUs, each of which is labeled with the size of the application data contained within (in bytes)
- *targetAudioBandwidth*: a target bandwidth for audio in kbps
- *pktHeaderLen*: the total amount of overhead present in each packet for headers at the transport, network and data link layers (above and beyond the actual data portion) in bytes.
- *totalBandwidth*: the available bandwidth for transmitting PDUs.

Output:

- a sequence of PDUs, representing a linear extension of the partial order over the PDUs implied by *G*, as per the stream abstraction defined in Section 2.3, with the following properties

  (1) When there are parallel stream objects with deliverable cells, and none of these stream objects is an audio element, cells are added to the linear extension by visiting the objects in round-robin order.

  (2) When one of the stream objects is an audio element[70], the audio element is given priority any time the total effective bandwidth for audio PDUs (counting only the application level bytes) falls below the *targetAudioBandwidth*.

Figure 6.4 Algorithm ReMDoR-LESTAB inputs, output

---

[70] Currently, ReMDoR does not permit multiple audio elements to be played simultaneously. Future work on ReMDoR may add *audio mixing*, to permit, for example, an audio track of background sounds (music, crowd noise, etc.) in parallel with an audio track of narration. This feature is interesting for evaluating PO/PR service since it motivates multiple priorities and/or levels of reliability for multiple audio streams.

```
Algorithm ReMDoR-LESTAB:
Pseudocode:
while not (front.isEmpty())
{
    vertex u = front.dequeue();
    vertex currentAudioElement = nil;
    while (u ≠nil) or (currentAudioElement ≠ nil)
    {
            if  (currentAudioElement ≠ nil) and the average bandwidth
            provided to audio up to this point < targetAudioBandwidth)
            {
                output the next PDU from currentAudioElement
                if (it the last PDU for currentAudioElement){ finish(audio);
            }
            }
            else
            {
                output the next PDU from the vertex u.
                if (that was the last PDU for u)
                {
                    finish(u)
                }
            }
    }
```

Figure 6.5 Algorithm ReMDoR-LESTAB Pseudocode

```
procedure finish(u)
foreach v in adj-list[u]   // (u,v) is an edge in G
    {
            if  (u == currentAudioElement)
                { currentAudioElement = nil;} // this audio element is finished
            if (in-degree[v] == 0)
               halt-with-error;
            // input was inconsistent, or graph contains a cycle
            decrement in-degree[v];  // logically, removes (u,v) from G;
            // optionally, we can also remove v from adj-list[x], but this is
            // unnecessary

            if in-degree[v] ==  0 // v can now be added to the sorted list
            {
                if (v is an audio element)
                    currentAudioElement = v;
                else
                        front.enqueue(v);
                }
    }
```

Figure 6.6 ReMDoR-LESTAB, implementation of procedure *finish()*

6.4 Verifying the sending order

In any realistic implementation of a PO/PR transport protocol, the available memory for buffering and resequencing out-of-order TPDUs at the partial order receiver is finite. If the PO receiver's buffers become full of packets that are undeliverable because their predecessors have not yet been received, a *resequencing deadlock* can occur. The resequencing deadlock problem for PO/PR protocols was first recognized in (Amer et al., 1994). That paper proposed the rule that POC users should submit TSDUs to the PO sender in a sequence that is a valid linear extension of the partial order. This section describes how the DAGITS algorithm is applied to enforce this initial sending order. We also present a more complete argument than the one in (Amer et al., 1994) justifying the need for enforcing this initial sending order in light of application-transport end-to-end flow control.

6.4.1 The ISOLE rule

As notation, we define the *ISOLE rule* as the rule that *the initial sending order for objects must be a linear extension of the partial order*. The ISOLE rule is enforced both for the submission of objects (TSDUs) by the application to the PO sender, and for the order of the *initial* transmission of TPDUs from the PO sender to the PO receiver.

6.4.2 Enforcement of the ISOLE rule in UTL PO/PR services

The following notes indicate how PO/PR services provided by UTL enforce the ISOLE rule:

- The POL layer implements a modification of the DAGITS algorithm (as explained below) to enforce the ISOLE rule for the order in which TSDUs are submitted via `utl_Write()` calls.

- All UTL middle layers (NUL, TOL, POL) enforce a rule that the order in which TSDUs are submitted by the layer above is the same order in which the corresponding TPDUs are submitted to the layer below. This enforces the ISOLE rule at the Service Access Point (SAP) between POL and the layer below.

- All UTL bottom layers (TXL, KXP, KX2, KX3) enforce a rule that the order in which TPDUs are placed in the sending window corresponds to the order in which the corresponding TSDUs were submitted by the layer above.

6.4.3 Using the DAGITS algorithm to enforce the ISOLE rule

To enforce the ISOLE rule for submitted TSDUs, a PO sender must comply with three constraints:

(1) Within each period, the order in which TSDUs are submitted must be a valid linear extension.

(2) No TSDU may be sumitted twice.

(3) All TSDUs of period *i* are submitted before any TSDUs of period *i*+1 are submitted.

Formally, we call this the ISOLE (Initial Sending Order must be a Linear Extension) problem, and formulate it as shown in Figure 6.7.

Problem ISOLE: Initial Sending Order must be a Linear Extension

Operations:

init(*G*) initialize data structures representing a directed acyclic graph *G*=(*V*,*E*) supplied by the caller.

returnCode = *submitTSDU*(*TSDU*, *objnum*) submit the TSDU with *objnum* as its object number; return *success* if TSDU was submitted, or *failure* to indicate that sending the TSDU would violate the ISOLE rule.

Figure 6.7 Problem ISOLE (operations)

The Algorithm PO-SENDER-ISOLE (PO Sender enforcement of ISOLE rule) enforces the above constraints by maintaining an adjacency list representation of the partial order, and a count of the number of submitted objects within the current period. Figure 6.8 shows a simplified version of the algorithm that assumes there is a single period. Extending this to multiple periods adds extra bookkeeping to the code, but does not change its running time.

In this algorithm, each object is initially marked as "not submitted yet". When a TSDU is submitted, a check is made to ensure all of the corresponding object's predecessors have already been submitted. This is done in $O(1)$ time by simply examining the in-degree of the object. Assuming that submission of the object is legal, the in-degree of the object's successors are then decremented, which takes $O(d)$ time, where d is the out-degree of the object. Thus the running time for each `Write()` operation is $O(d)$, and overall, the algorithm takes time $O(n+e)$.

6.5 Resequencing out-of-sequence PDUs for delivery using partial order

At the PO receiver, out-of-order PDUs must be resequenced for delivery. Early specifications of POC accomplished this using a matrix representation of the partial order, where $A[i,j]$ indicated $i \prec j$ in the partial order.

The matrix approach to resequencing PDUs for PO service

Using the matrix approach, each time a packet $[i]$ was delivered, all elements in column i would be cleared, thus removing the constraint on the successor objects. The operation that delivers objects would simply scan the buffered objects to

```
    Algorithm PO-SENDER-ISOLE
    Data Structures:
            boolean:        submitted[i]        // has TSDU [i] been submitted
                                                // yet? Initialized to false
                int:        in-degree[i]        // current in-degree of vertex [i]
    list of vertices:         adj-list[i]       // list of outgoing edges from
                                                vertex [i]

    operation init(G); // initialize submitted, in-degree and adj-list from G.
    boolean submit_TSDU (unsigned int objNum); // called from utl_Write()
         // objNum is the object number of the submitted TSDU
         // return value is true on success, and false if an error occurs.

operation submitTSDU(objNum):
         if (submitted(objNum))
           return false; // this object was already submitted
         if (in-degree(objNum)>0)
           return false; // sending this object would violate the partial order
         // now we know that sending this object will be legal
         send the TSDU over the network as a TPDU;
         foreach v in adj-list[objNum]   // (u,v) is an edge in G
         {
              decrement in-degree[v];  // logically, removes (u,v) from G;
         }
         return true;
```

Figure 6.8 Algorithm PO-SENDER-ISOLE

see if any had a row that was entirely filled with zeros; these objects were deliverable. This algorithm requires at least[71] time $\Omega(n)$ for each packet delivery, and thus time $\Omega(n^2)$ for the entire algorithm. This analysis raises two questions:

(1) Can we do better using an adjacency list representation? (as in the DAGITS algorithm)

(2) Does it really matter? What is considered a reasonable amount of processing in a transport layer protocol?

The matrix approach is not acceptable for PO service

Given that TCP uses a linear search for packet resequencing, one might be tempted to conclude that the matrix-based approach to processing a partial order is perfectly reasonable. After all, if TCP can resequence packets using a linear search, why go to the trouble to eliminate an $O(n)$ operation on each packet arrival or delivery in POC? However, this reasoning has several flaws.

First, in comparing the resequencing algorithms for a PO protocol to those of an ordered protocol (e.g., TCP), one must distinguish between a linear search of k elements (the number of out-of-sequence elements currently buffered) vs. a linear search of a data structure containing n elements, where n is the number of elements in the partial order.

Second, the $O(k)$ operation in TCP is the *worst-case* upper bound of the *uncommon* case, while the $O(n)$ `Read()` and `Write()` operations in the matrix-based version of POC are in fact $\Theta(n)$; that is, *every* invocation of these operations requires no more—and no less—than a linear amount of time.[72]

---

[71] Big-Omega of n ($\Omega(n)$) is used here to indicate that this is a lower-bound, not an upper bound on the running time.

[72] $\Theta(n)$ indicates that the running time is bounded above by $O(n)$ and below by $\Omega(n)$.

Finally, it is quite reasonable for values of n to reach into the hundreds for the size of a partial order (see the example documents in the appendix.) Meanwhile the number of outstanding out-of-order packets in a TCP connection is limited not only by the maximum window size, but also by the congestion window. It is true that higher bandwidths, and longer delay paths are leading to larger window sizes (see, for example, RFC1323). We have no hard data regarding the actual distribution of effective window sizes in the Internet, however, our anecdotal evidence suggestions that TCP congestion window sizes $\ll 50$ packets are still the norm. (A more scientific measurement of what is "normal" here would be a good subject for future investigation, perhaps by measuring window sizes seen on a busy web/mail/telnet/ftp server.) Therefore, to avoid having to do $\Theta(n)$ work to resequence out-of-order PDUs on each PDU arrival, we propose as an alternative to the matrix approach, an adaptation of the DAGITS algorithm shown in Figures 6.9 through 6.13. We call this algorithm PO/R-RCV-RESEQ (PO/R receiver resequencing).

Incorporating explicit release synchronization into PO resequencing

The structure of the pseudocode for Algorithm PO/R-RCV-RESEQ justifies the claim that explicit release synchronization adds little complication to PO/R resequencing. In fact, all that is necessary to implement explicit release is (1) remove the call to the *releaseSuccessors*() procedure from the implementation of *getNextTSDU*(), and (2) make the local procedure *releaseSuccessors*() an operation that can be invoked directly by the transport service user. (Incorporating explicit release into the PO/PR algorithm is more difficult, but still feasible, as Section 6.8 will illustrate.)

```
Algorithm PO/R-RCV-RESEQ:
Data Structures:
          int:          n           // number of elements in PO
          int:        count         // number of TSDUs delivered so far
  boolean:   received[i]   // has TSDU [i] been submitted yet? Initialized to false
TPDU pointer: data[i]           // pointer to data; initialized to nil
         int:  in-degree[i]  // current in-degree of vertex [i]
  list of vertices: adj-list[i] // list of outgoing edges from vertex [i]
  queue of vertices: output     // queue of vertices with in-degree 0, and
                                // data pointer not equal to nil.

Operations:
  operation init(G);          // initialize values from graph G.
  operation processIncomingTPDU(TPDU, objnum); // arrival from network
  operation getNextTSDU() returns TSDU; // returns next TSDU in PO
                                 // blocks until one can be delivered; returns nil on EOF
  operation isAnythingDeliverable() returns boolean;
                                // true if something can be delivered, or if we
                                // have reached EOF.  Signifies that a call to
                                // getNextTSDU() will not block if called.
Local Procedures:
  local procedure releaseSuccessors(u);      // release successors of object u
```

Figure 6.9   Algorithm PO/R-RCV-RESEQ, specification

```
Algorithm PO/R-RCV-RESEQ:
operation processIncomingTPDU(TPDU, objNum):
    if (received(objNum))
      discard TPDU; // this object is a duplicate (perhaps a retransmission)
    else
            {
            data[objNum] = TPDU;
            if (in-degree(objNum) == 0)
```

```
                output.enqueue(v); // queue this object for delivery.
            }
    return;
```

Figure 6.10 PO/R-RCV-RESEQ, operation *processIncomingTPDU()*

```
operation getNextTSDU() returns TSDU:
    local variable TPDU pointer tpdu;
    if (count == n) return nil; // all objects have been delivered
    wait (not output.empty());// wait until something is added to output queue.
    tpdu = output.dequeue;
    releaseSuccessors[tpdu.objnum];
            // remove this line to provide explicit release
    return encapsulated TSDU from inside tpdu
```

Figure 6.11 PO/R-RCV-RESEQ, operation *getNextTPDU()*

```
operation isAnythingDeliverable() returns boolean:
    return (count == n or not output.empty());
```

Figure 6.12 PO/R-RCV-RESEQ, operation *isAnythingDeliverable()*

```
local procedure releaseSuccessors (u):
    foreach v in adj-list[u]  // (u,v) is an edge in G
    {
            decrement in-degree[v];  // logically, removes (u,v) from G;
            if (in-degree[v] ==  0 and received[v]) // v is now deliverable
                output.enqueue(v);
    }
```

Figure 6.13 PO/R-RCV-RESEQ, procedure *releaseSuccessors()*

6.6 Implementing the PR semantics of POCv2

Modifying Algorithm PO/R-RCV-RESEQ to implement the PR reliability class of POCv2 presents a particularly interesting challenge. Recall that the semantics of U and PR objects in POCv2 allow the application to say about a particular object:

- An object in class U or class PR is useful but not essential, and the delivery of other objects (regardless of their reliability class) should not be delayed by its absence.
- A class U object should never be retransmitted.
- A class PR object is important enough that it should be retransmitted if extra time is available.

To implement these semantics, any time the application invokes the transport service's `Read()` operation, if there is no data currently deliverable, the transport layer must be able to efficiently answer the question: "Is there some object *x* that would become deliverable if a non-empty set of undelivered PR and/or U objects were declared lost?" If the answer is yes, then we say that object *x* is *waiting,* and that the undelivered U and PR predecessors of *x* are *loss candidates.* The remainder of this section presents formal definitions for these notions; these definitions are used in Sections 6.7 and 6.8 to develop algorithms for PO/PR service.

Definition 6.6.1: An object *x* is ***resolved*** if either:

- x has been delivered and its successors have been released, *or*
- *x* has been declared lost and its successors have been released.

(Object *x* is *unresolved* if it is not *resolved*.) ❑

The terms *resolved* and *unresolved* allow us to avoid the awkward phrase "delivered or declared lost". The term *resolved* also encapsulates the *releaseSuccessors*() operation together with that of delivering an object or declaring it lost; this encapsulation will be particularly useful when moving from the basic algorithm without explicit release to the more advanced algorithm that incorporates explicit release.

The semantics of POCv2 require that the receiver does not declare anything lost until both of the following are true:

(1) the service user is currently issuing a read request,

(2) nothing is currently deliverable, but something would *become* deliverable if one or more objects were declared lost.

These two conditions motivate the definition of a *waiting* object:

Definition 6.6.2: An object *x* is ***waiting*** if and only if all of the following are true:

(1) *x* is buffered

(2) *x* has at least one unresolved immediate predecessor, and

(3) *x* has no unresolved reliable proper predecessors. ❑

The concept of a waiting object is useful because the presence of one or more waiting objects triggers the POCv2 receiver's *getNextTSDU*() operation to declare objects lost. However, it is not enough to simply define the concept of a waiting object; we also require an efficient algorithm to determine which objects are waiting at any point in time.

To accomplish this, one suggestion would be to maintain in each object *x*, a count of the number of unresolved reliable predecessors of object *x*. This would allow us to evaluate condition (3) of the definition of *waiting* in constant time. Conditions (1) and (2) can already be evaluated in constant time:

- Condition (1) requires a per-object boolean variable called *buffered*(*i*) to indicate whether or not the object is buffered. This boolean variable is initialized to **false** during the *init*(*G*) operation, and is modified whenever an object is placed in or removed from the buffer.
- Condition (2) requires us to check the in-degree of the object. The *in-degree*(*i*) variable is initialized from the service profile, and is decremented each time a covered object releases its successors.

Thus both the evaluation of conditions (1) and (2), and the maintenance of the necessary state (the variables *buffered*(*i*) and *in-degree*(*i*)) requires only constant time per protocol operation.

Being able to evaluate all three conditions in constant time allows us to maintain a list of all objects that meet the three conditions by simply checking all three conditions whenever any operation is performed on an object that could change any of the three conditions. Since the previous algorithm already keeps the state necessary to check conditions (1) and (2) in constant time, the only additional processing needed is the maintenance in each object of the number of unresolved reliable predecessors.

However, keeping track of the number of unresolved reliable predecessors is more information than we need, and would likely require too much work. Setting aside for the moment the issue of how such a value would be initialized, consider just the problem of maintaining the value. Suppose that the partial order is an antichain of

$n/2$ reliable objects followed by a chain of $n/2$ unreliable objects. Each time a reliable object is resolved, this action could affect the number of unresolved reliable predecessors of $\Theta(n)$ objects, and there are $\Theta(n)$ such reliable objects. Therefore this example will require $O(n^2)$ processing (which is especially poignant, since in this special case, $O(n+e) \sim O(n)$.) However, as we explain below, we can do better.

What we would prefer is a method that allows us to keep track of just enough information to determine in constant time whether an object meets condition (3) of the definition of *waiting*, and we would like to be able to determine this with processing that adds no more than $O(n+e)$ to our running time for processing an entire period of n objects. Even if this time is distributed unevenly among the operations, if we can bound the total time, we can use an amortized analysis to argue that the total running time is not increased. It would also be particularly convenient if the expensive operations did *not* take place during the routine that handles incoming packets. The real-time performance of the incoming packet processing is crucial, since slow processing can lead to overflows in the incoming packet queue, and consequently to packet loss.

A key observation is that condition (3) of the definition of waiting requires only a binary decision: either the number of unresolved reliable predecessors of an object x is greater than zero, or it is not. One way to determine this is to keep track of how many of the *immediate* predecessors of x (also known as the covered objects of x) have unresolved reliable predecessors. If none of the immediate predecessors of object x have unresolved reliable predecessors, then by a simple argument (made formally in Theorem 6.6.5 below), neither does object x. This observation motivates the following definitions:

Definition 6.6.3: *COURPs*(y) is the set of **C**overed **O**bjects with **U**nresolved **R**eliable **P**redecessors, and consists of all objects *x* such that

- *x* is covered by *y* (i.e., *y* covers *x*, or equivalently, *x* is an immediate predecessor of *y*; see definition of *covers* in Chapter 2) and,
- *x* has at least one reliable predecessor that is unresolved. (Note that this predecessor need not be a proper predecessor. Just as the notions of ancestor and descendent are often treated as reflexive binary relations over the nodes in a tree, with node *x* being both an "ancestor" and a "descendant" of itself, the predecessor relation is also defined as reflexive. An element *y* of a partial order *PO* is a predecessor of *x* if and only if $y \prec= x$ w.r.t *PO*. Thus if *x* has reliability class R, it may be considered a reliable predecessor of itself.) ❑

Definition 6.6.4: *numCOURPs*(y) = |*COURPs*(y)|, that is, the number of objects covered by *y* that have at least one unresolved reliable predecessor. ❑

The advantage of tracking *numCOURPs*(*x*) rather than the number of unresolved reliable predecessors is this: tracking *numCOURPs*(*x*) requires the algorithm to keep only local information in each node, which is less expensive than maintaining global information in each node. Essentially, the algorithm exploits the transitivity of the partial order to save computation cost.

The following Theorem about *numCOURPs* will be useful, since it shows that the predicate (*numCOURPs*(*x*) == 0) is equivalent to one part of the definition of a *waiting* object (Definition 6.6.2).

Theorem 6.6.5: For any object *x*:

(*numCOURPs*(*x*) == 0) ⇔ (*x* has no unresolved reliable proper predecessors)

Proof: (⇒, by contradiction.) Let *a* be an unresolved reliable proper predecessor of *x*, and let *coveredBy*(*x*) be the set of all objects covered by *x* (*x*'s immediate proper predecessors). Since object a is a proper predecessor of x, it must be a predecessor (not necessarily proper) of some element w in the set *coveredBy*(x). However, this would imply that w is a covered object of x with an unreleased reliable predecessor, namely a, contradicting the assumption that *numCOURPs*(x) == 0. ❑

(⇐) (By contradiction) Suppose that x has no unresolved reliable proper predecessors, but that, nevertheless, *numCOURPs*(x) > 0. This would imply that there exists some $w \in$ *coveredBy*(x) such that w has an unresolved reliable predecessor. However, since $w \prec x$, any unresolved reliable predecessor of w would necessarily be an unresolved reliable proper predecessor of x, contradicting our premise. Therefore, no such w exists, and consequently (*numCOURPs*(x) == 0). ❑

In the case where the POCv2 receiver has no deliverable data, but has at least one waiting item that could be delivered if its loseable predecessors were declared lost, the receiver will need an algorithm to actually *find* all of these unresolved unreliable predecessors. Furthermore, to preserve the partial order, these objects will need to be resolved in some linear extension of that partial order, which will require a topological sort. Therefore, the discussion of POCv2-style partial reliability in Section 6.7 includes an algorithm to find these unresolved unreliable predecessors and resolve them in linear extension order. The following definition is useful in the discussion of that algorithm:

Definition 6.6.6: Given that y is a waiting object, the set $L(y)$ consists of all of y's undelivered unreliable predecessors. The elements of $L(y)$ are referred to as *loss candidates.* ❑

The capital letter "*L*" in the notation *L*(*y*) stands for "loss candidates", but it can also serve as a reminder that all objects in the set *L*(*y*) are *loseable* objects (objects with reliability class U or PR); indeed, these are exactly the set of objects that must be resolved before *y* may be delivered. It is useful to note, however, that not all of the objects in *L*(*y*) are *necessarily* ones that will be declared lost; it is possible within the constraints of the definition that some or perhaps even all of the elements of *L*(*y*) may have actually arrived, and will be *delivered* rather than being declared lost.

With this framework in place, we can now proceed to the development of algorithms for the PO/PR receiver

6.7 PO/PR-DEL-BASIC: Basic POCv2 delivery (no stream, no explicit release)

If the PR semantics of POCv2 are to be feasible, we need an algorithm that can either:

(1) determine that no waiting objects currently exist, or

(2) identify at least one waiting object *x*, and the set of loss candidate objects *L*(*x*) that must be resolved before that object *x* can be delivered.

Furthermore, we need to be able to invoke this algorithm efficiently at any time the application performs a `Read()` operation on an empty input queue. Our strategy is to do enough bookkeeping with every packet arrival event, delivery event, and release successors event, such that the total running time is still O(*n*+*e*) for the processing of an entire period of objects.

In this section, we develop and present a basic algorithm for PO/PR delivery (PO/PR-DEL-BASIC) that follows the PR semantics of POCv2. The PO/PR-

DEL-BASIC algorithm provides the reader with a clear explanation of the extra processing necessary to support the POCv2 PR semantics without the clutter introduced by supporting explicit release and the stream abstraction. PO/PR-DEL-BASIC then serves as the basis for Algorithm PO/PR-DEL-FULL, which implements the full specification of POCv2, including the explicit release mechanism and stream abstractions

Overview of Section 6.7

Section 6.7.1 develops the PO/PR-DEL-BASIC algorithm by presenting pseudocode for the basic operations and local procedures. In each case, we take the corresponding operations from the PO/R algorithm PO/R-RCV-RESEQ and explain what changes are necessary to incorporate the U and PR classes of POCv2. Section 6.7.2 then provides proofs of correctness and running time for this pseudocode, and elaborates some of the details omitted in the pseudocode version. Section 6.7.3 provides a comparison and contrast between the PO/R-RCV-RESEQ and PO/PR-DEL-BASIC algorithms. Section 6.7.4 concludes with a summary of the main points related to the PO/PR-DEL-BASIC algorithm.

6.7.1 Extending the PO/R-RCV-RESEQ algorithm to incorporate PR

We begin our presentation of PO/PR-DEL-BASIC with an outline of the basic operations and local procedures (Figures 6.14 through 6.21).

Comparing Figure 6.14 with Figure 6.9, we see that the basic operations in PO/PR-DEL-BASIC are exactly the same as those in PO/R-RCV-RESEQ. We retain the local procedure *releaseSuccessors*() (Figure 6.16), and in addition, we have added two new local procedures that compute the *numCOURPs* and $L(x)$ values defined previously (Section 6.6). We now show how the pseudocode for each of these four operations is extended to incorporate partial reliability.

Compare the implementation of the *isAnythingDeliverable*() operation for the PO/R-RCV-RESEQ algorithm (Figure 6.12) with the implementation of the isAnythingDeliverable operation for PO/PR-DEL-BASIC shown in Figure 6.15. For PO/R service, this algorithm is $O(1)$, and trivial to implement. However, in PO/PR, the processing is more complex. Figure 6.15 shows a partially specified pseudocode for *isAnythingDeliverable*()—partially specified, in the sense that at this stage, the pseudocode does not explain how to compute the value of the boolean expressions in the second and third **if**-tests; later sections explain how this can be done in constant time.

Figure 6.16 provides an outline of the pseudocode for the *getNextTSDU*() operation for PO/PR service. If we compare this pseudocode to that of the *getNextTSDU*() operation for PO/R service, we first note that the **if**-block following the **wait** statement is exactly equivalent to the implementation of the *getNextTSDU*() operation for PO/R service. Therefore, it is the first **else**-block that is of interest. In Section 6.7.2, we will prove that whenever we reach this block, there will always be at least one waiting item, and that we can find one of these waiting items in $O(1)$ time (given some prior $O(n+e)$ processing; this prior processing gets amortized in with the $O(n+e)$ processing of Algorithm PO/R-RCV-RESEQ). We will then show how the procedure *fillDeliverOrDeclareLostQueueWithSortedLSet*(y) can find all the loseable items that need to be declared lost (or delivered, if by chance they arrive in time) so that the waiting item becomes deliverable. This procedure not only finds these items, but also topologically sorts them, and places them into a queue called the *deliverOrDeclareLost* queue. We will show that all of the processing that takes place in this procedure over the lifetime of the connection cannot exceed $O(n+e)$, by a simple accounting argument that no node or edge can be processed by this algorithm more than once. We will also show that once this queue has been constructed for any given waiting item, we can march through this queue declaring items lost until the waiting item comes to the front of the queue, at which point it can be delivered as the next TSDU.

The *processIncomingTSDU*() operation is virtually unchanged; the only difference is that we may have to remove an item from the *deliverOrDeclareLost* queue if it becomes deliverable while it is sitting in that queue. (Note that because of this processing, the *deliverOrDeclareLost* queue is not strictly a queue, since a deliverable item that is removed from the *deliverOrDeclareLost* queue by the *processIncomingTPDU*() procedure may in fact be found somewhere other than the head of this queue.)

Finally, we turn to the *init*(G) operation. We did not show pseudocode for this operation in the case of the PO/R-RCV-RESEQ algorithm (6.5.1) because it was trivial to implement; however for the PO/PR-DEL-BASIC algorithm, there is an additional step: the updating of the *numsCOURPs* values for each node. This updating is done by a truncated DFS operation called *updateNumCOURPsofSuccessorNodes*(*x*) that starts from each element that has no predecessors in the graph, and stops when a reliable element is encountered, or an element is encountered that already has *numCOURPs*= =0. The DFS is resumed each time a reliable node is delivered; in this manner, over the course of the algorithm, each node's *numCOURPs* is updated, and each edge in the graph is traversed at most once. Section 6.7.2 will explore further how this processing works, and will prove that the total running time for this processing does not exceed $O(n+e)$ over the course of the algorithm.

```
Algorithm PO/PR-DELBASIC:
Data Structures:
        int:        n           // number of elements in PO
        int:      count         // number of TSDUs delivered so far
  boolean:    arrived[i]        // has TSDU [i] arrived yet? Initialized to false
  boolean:   resolved[i]        // has TSDU [i] been resolved yet? Initialized to false
  enum:        color[i]:        {white, gray, black}        // marks nodes in DFS
  TPDU pointer:
                 data[i]        // pointer to data; initialized to nil
        int:  in-degree[i]      // current in-degree of vertex [i]
  list of vertices: adj-list[i] // list of outgoing edges from vertex [i]
  list of vertices: trpot-adj-list[i] // adjacency list in transpose of partial order
  queue of vertices: output     // queue of vertices with in-degree 0.
  list of vertices: deliverOrDeclareLost        // loss candidates to be resolved

Operations:
  operation init(G);          // initialize values from graph G.
  operation processIncomingTPDU(TPDU, objnum); // arrival from network
  operation getNextTSDU() returns TSDU; // returns next TSDU in PO
                               // blocks until one can be delivered; returns nil on EOF
  operation isAnythingDeliverable() returns boolean;
                              // true if something can be delivered, or if we
                              // have reached EOF.  Signifies that a call to
                              // getNextTSDU() will not block if called.
Local Procedures:
  local procedure releaseSuccessors (u);      // release successors of object u
  local procedure updateNumCOURPsofSuccessorNodes(x);
            // called whenever we deliver a reliable object; this procedure and its
            // recursive calls on loseable successors help to find waiting objects
  local procedure fillDeliverOrDeclareLostQueueWithSortedLSet (x);
            // called on a waiting object to find the set L(x); recursive calls
            // constitute a DFS of the transpose of the PO, which is a topological sort.
```

Figure 6.14 Algorithm PO/PR-DEL-BASIC, specification

```
operation isAnythingDeliverable() returns boolean:
{
    if (not output.empty() or count == n) return (true);
    else if (there are no buffered items)
```

```
        {return false} // there's no data at all
    else if (there exist no waiting items) // see defn 6.6.2; thm 6.7.1
        {return false} // there's data, but each object is waiting on a least one
        // reliable predecessor, so declaring things lost won't help
    else {return true} // there is at least one item we could deliver if we declared
        // its predecessors lost; see theorem 6.7.8
```

Figure 6.15 PO/PR-DEL-BASIC, operation *isAnythingDeliverable()*

```
operation getNextTSDU() returns TSDU:
    local variable TPDU pointer tpdu;
    if (count == n) return nil; // all objects have been delivered
    wait(isAnythingDeliverable);
        // if false, sleep; recheck condition after each processIncomingTPDU() call;
        // in practice, the sleep can be avoided by never calling getNextTSDU()
        // without first checking isAnythingDeliverable().
    if (not output.isEmpty())
        {
            tpdu = output.dequeue;
            releaseSuccessors(tpdu.objnum);
            return encapsulated TSDU from inside tpdu
        }
    else // deliver a waiting item (after possibly declaring some items lost)
        {
        if (deliverOrDeclareLost.isEmpty())
        {
            choose y = an arbitrary waiting item        // see Thm. 6.7.9
            fillDeliverOrDeclareLostQueueWithSortedLSet(y);
            // find the set L(y), consisting of all of y's undelivered unreliable
            // predecessors, and topologically sort them according to the
            // partial order, and place these items on the
            // deliverOrDeclareLost queue.
        }
        while (true)
        {
            if (not output.isEmpty()) // if a waiting object become deliverable
            {
                x = output.remove;
                releaseSuccessors(x.objnum);
                return encapsulated TSDU from inside x
                }
                else
                {
                      x = deliverOrDeclareLost.remove();
                      declareLost(x);
                      releaseSuccessors(x.objnum);
                }
          }
assert(false); // by theorem 6.7.9, we should never reach this statement
```

Figure 6.16 **PO/PR-DEL-BASIC**, **operation** *getNextTSDU**()***

```
operation processIncomingTPDU(TPDU, objNum):
    if (received(objNum))
       discard TPDU; // this object is a duplicate (perhaps a retransmission)
    data[objNum] = TPDU;
    if (in-degree(objNum) == 0)
    {
            if (objNum is in the deliverOrDeclareLost queue)
            {
                  deliverOrDeclareLost.removeByObjNum(objnum);
            }
            // because of the preceding line, deliverOrDeclareLost is not strictly a
            // queue
            output.enqueue(objNum); // queue this object for delivery.
    }
    return;
```

Figure 6.17 **PO/PR-DEL-BASIC, operation** *processIncomingTPDU()*

```
operation init(G):
    foreach (node x in G)
```

```
{
        initialize numCOURPs(x) to in-degree(x);
            // G' is initially a copy of G
        initialize color(x) to white; // used in DFS algorithm; see Fig. 6.21
};
foreach (node x that is unreliable, and has numCOURPs(x)= = 0)
{
        updateNumCOURPsofSuccessorNodes(x);
        // logically, remove x from the graph G'
}
foreach (node x in G)
{
        traverse x's adjacency list, constructing a corresponding
        adjacency list trpot-adj-list(y) for every y in the transpose of G,
        i.e., $TRPO^T$.
}
return;
```

Figure 6.18 PO/PR-DEL-BASIC, operation *init()*

```
local procedure updateNumCOURPsofSuccessorNodes(x):
    foreach (y in successors(x))
        {
            decrement numCOURPs(y)
            if (numCOURPs(y) == 0 and y is not reliable)
                updateNumCOURPsofSuccessorNodes(y);
        }
    }
    return;
```

Figure 6.19 PO/PR-DEL-BASIC, *updateNumCOURPsofSuccessorNodes()*

```
local procedure releaseSuccessors (u):
    foreach v in adj-list[u]  // (u,v) is an edge in G
    {
            decrement in-degree[v];  // logically, removes (u,v) from G;
            if (in-degree[v] ==  0 and received[v]) // v is now deliverable
            {
                if (object objNum is in the deliverOrDeclareLost queue)
                {
                    deliverOrDeclareLost.removeByObjNum(objnum);
                } // deliverOrDeclareLost is not strictly a queue
                output.enqueue(v);
            }
            if (object u is reliable)
            {
                decrement numCOURPs(v); // u was a covered object with an
                                // unresolved reliable (not proper) predecessor.
                if (numCOURPs[v] ==  0)
                    updateNumCOURPsofSuccessorNodes(v);
                    // propagate the effect
            }
    }
```

Figure 6.20 PO/PR-DEL-BASIC, procedure *releaseSuccessors()*

```
local procedure fillDeliverOrDeclareLostQueueWithSortedLSet (x):
// compare with DFS of CLR, p. 478.  As in CLR's version, colors indicate
// discovery/finishing times:
//    white=undiscovered, gray=discovered, not finished; black=finished
    color[x] = gray; // as in DFS of CLR (p. 478),.
    if (in-degree[x]= =0) // can only be true on recursive call; original call is
            // always on a waiting object, which cannot have in-degree= =0.
            {
                color[x] = black; // finished with x
assert (not buffered(x)); // if it were, we shouldn’t be here; this routine
                // would not be invoked if a deliverable object existed.
                deliverOrDeclareLost.enqueue(x);
                return;
            }
    // note: the check on in-degree= =0 is not redundant, since we do not
    // actually remove the edges from trpot-adj-list as in-degree is decremented.
    foreach v in trpot-adj-list[w]
            // (v,w) is an edge in G, (w,v) is an edge in G′,
    {
            if (resolved[v])
                {continue};
            // resolved elements treated as black, already finished;
            else if (color[v]= =white)
                { fillDeliverOrDeclareLostQueueWithSortedLSet (v);}
                                                  // finished with v
    }
    color[w] = black;
    deliverOrDeclareLost.enqueue(x);
    return;
```

Figure 6.21 PO/PR-DEL-BASIC, *fillDeliverOrDeclareLostQueueWithSortedLSet(x)*

6.7.2 Proofs of correctness, running time for PO/PR-DEL-BASIC pseudocode

Our series of proofs is divided into two parts. In the first part, we concentrate on the correctness of the *isAnythingDeliverable*() routine, which is dependent on the maintenance of the list of waiting items, which in turn depends on maintaining the *numCOURPs* values for each object. In the second part, we focus on the running time of the DFS of the $TRPO^{T}$ to compute the $L(y)$ set and the correctness of the *getNextTSDU*() operation. We omit formal proofs of the correctness and running time of the remaining routines, since they are either self-evident, or follow directly from the remaining material; their inclusion would not shed any extra light.

Proofs related to maintaining the *numCOURPs* values.

Theorem 6.7.1: The PO/PR receiver can maintain a list of all waiting objects by adding extra processing to the *init*(), *getNextTSDU*(), and *processIncomingTPDU*() operations of Algorithm PO/R-RCV-RESEQ, without increasing the total running time of $O(n+e)$ for the processing of an entire period of n objects.

Proof: We divide the proof into three parts:

(1) We can initialize the correct value of *numCOURPs* for each object by adding code to the *init*() operation.

(2) We can maintain the correct value for *numCOURPs*, and maintain a list of *waiting* items by adding code to the *getNextTSDU*() operation.

(3) The running time of the added code is $O(n+e)$ over a sequence of n *getNextTSDU*() and n *processIncomingTPDU*() operations.

Part (1) of proof: Our claim is that after the initial call to *init*(G), the *numCOURPs* value for each x is correct. Consider the set *min*(G) consisting of all elements in G that have *in-degree* 0; that is, the minimum elements in the partial order. By transitivity, all elements of G contain at least one (not necessarily proper) predecessor in the set *min*(G). The operation *updateNumCOURPsofSuccessorNodes* is called on every element of *min*(G), essentially simulating the resolution and removal from G of all unreliable objects with no unresolved proper predecessors. Let G' be the graph consisting of G minus all elements that are "removed" by the operation *updateNumCOURPsofSuccessorNodes*. After the removal of all the unreliable elements from G' that have no reliable predecessors, the in-degree of the remaining graph (which corresponds to the value of *numCOURPs*) will indicate only the number of covered objects of each x in G that were not able to be removed because they have at least one reliable predecessor. Therefore, after the *init*(G) operation, the values of *numCOURPs* are correct.

Part (2) of proof: Our claim is that the correctness of the value *numCOURPs* is maintained throughout the algorithm. Note that the only action that can change the value of numCOURPs is the resolution of a reliable object. Reliable objects can only be resolved by delivery, and in this version of the algorithm, each time an object is delivered, the procedure *releaseSuccessors*() is immediately called. For each reliable object, *releaseSuccessors*(x) calls *updateNumCOURPsofSuccessorNodes*(x), which essentially removes element x from the graph G' just as in the proof of part (1) above. Thus the correctness of the *numCOURPs* value is maintained throughout the course of the algorithm.

Part (3) of proof: Our claim is that the asymptotic worst-case running time of the extra processing added by the invocations of *updateNumCOURPsofSuccessorNodes*() in both the *init*() and *releaseSuccessors*() operations is $O(n+e)$. This property follows immediately from the observation that the *updateNumCOURPsofSuccessorNodes*()

calls are essentially nothing more than a shadow version of the basic algorithm used for resequencing and delivering objects. Since this processing is equivalent to the processing that is done when each object is either delivered or declared lost—foreshadowing this processing for unreliable objects, and executing in parallel to this processing for reliable objects—the asymptotic running time is the same. We can amortize the extra processing of *updateNumCOURPsofSuccessorNodes*() by charging the time consumed to the actual resolution of each object when it is delivered or declared lost. Therefore, from an amortized sense, the total running time for a period of *n* objects with *e* edges is not increased by the extra processing needed to maintain the value *numCOURPs*. ❑.

Lemma 6.7.2: Given the index of any waiting object *b*, a topological sorting of the set *L*(*b*) can be produced by the reverse finishing times of a DFS of the unresolved elements of the $TRPO^{\mathrm{T}}$

$$\left\langle x_1, x_2, \cdots, x_{|L(b)|-1}, b \right\rangle$$

where the last element in the topological sorting of *L*(*b*) is the element *b* itself.

Proof of Lemma 6.7.2: Consider a DFS that follows the $TRPO^{\mathrm{T}}$ from element *b*, but prunes the DFS search-tree whenever an element is found that has zero unresolved predecessors. The lemma makes three claims:

(1) that all unresolved predecessors of *b* will be located by this method,

(2) that the DFS can put them in topologically sorted order, and

(3) that the element *b* will end up at the end of the list.

To prove claim (1) by contradiction, assume that some object *x* is an unresolved predecessor of *b* that is not located in the DFS search tree described above. This assumption implies that there are two possible cases for object *x*, both of which lead to a contradiction:

- Case(i): Object *x* cannot be reached by traversing the edges of TRPO^T from *b* to *x*, which would imply that it cannot be reached by traversing the edges of TRPO from *x* to *b*, contradicting the assumption that *x* is a predecessor of *b*.
- Case (ii): Object *x* can be reached by traversing the edges of TRPO^T from *b* to *x*, but some element *y* with zero unresolved predecessors lies on the path between *b* and *x*, and consequently, the DFS search tree is pruned before reaching *x*. However, if element *y* lies on the path in TRPO^T from *b* to *x*, then *x* is an unresolved predecessor of *y*. Hence, no such *y* can exist.

With respect to claim (2), it is well known that a topological sort of a directed graph can be obtained from the reverse finishing times of a DFS; see for example, Theorem 23.11 of (Cormen, Leiserson and Rivest, 1990; hereafter referred to as CLR.) Equivalently, the unreversed finishing times of a DFS of TRPO^T will also give us a topological sort (if $v \prec u$ in TRPO, then $u \prec v$ in TRPO^T, thus by Theorem 23.11 of CLR, $\mathit{finishingTime}[v]<\mathit{finishingTime}[u]$ in the DFS.). Therefore, by adding the nodes in the DFS to a list as they are finished, the topological sorting can be computed without additional expense. Claim (3) follows immediately from the proof of claim (2), since object b would be the last to be finished in the DFS search tree. ❑

Lemma 6.7.3: The procedure *fillDeliverOrDeclareLostQueueWithSortedLSet* of Figure 6.21 carries out the topological sort described in Lemma 6.7.3.

Proof of Lemma 6.7.3: Follows immediately from comparison of code with DFS-VISIT(*u*) of CLR, p. 478. ❑

Lemma 6.7.4: The first element x_1 in a topological sorting of *L*(*b*)

$$\left\langle x_1, x_2, \cdots, x_{|L(b)|-1}, b \right\rangle$$

is guaranteed to have no unresolved predecessors, and therefore can be resolved immediately.

Proof of Lemma 6.7.4: The DFS used to construct this sequence bottoms out only when an element is encountered that has no unresolved predecessors. Therefore, the first element to finish is guaranteed to have no unresolved predecessors. If it has no unresolved predecessors, it can be declared lost or delivered immediately.❑

Lemma 6.7.5: For each item x_i in the topological sorting of *L*(*b*), all elements $x_j \in L(x_i), j \neq i$, will precede x_i in the sequence.

Proof of Lemma 6.7.5: Since *L*(*b*) consists of all unresolved predecessors of *b*, and since $x_j \in L(b)$, all elements $x_j \in L(x_i)$ must also be in *L*(*b*). If some element $x_j \in L(x_i), j \neq i$, then x_j precedes x_i in the partial order. Therefore x_j must also precede x_i in any topological sorting based on that partial order. ❑

Theorem 6.7.6: Let *b* be a waiting object. The following pseudocode will correctly resolve all elements in *L*(*b*), ultimately resulting in the delivery of object *b*:

```
construct the sequence $\left\langle x_1, x_2, \cdots, x_{|L(b)|-1}, b \right\rangle$ as per Lemma 6.7.2
for (i=1; i<=|L(b)|; i++)
    {resolve ($x_i$)};
```

Proof of Theorem 6.7.6: By Lemma 6.7.4, the first element x_i of this topological sorting can be resolved immediately. By Lemma 6.7.5, all the unresolved predecessors of each element x_i precede element x_i; therefore, after each element x_i is resolved, element x_{i+1} will have no unresolved predecessors. By induction, therefore, we can resolve all elements of $L(b)$ by proceeding through the topological sorting of $L(b)$, ending with the resolution of object b. Since object b is buffered, this resolution will consist of the delivery of object b. ❑

Theorem 6.7.7: The PO/PR receiver pseudocode shown in Figure 6.15 correctly implements the *isAnythingDeliverable*() operation.

Proof of Theorem 6.7.7: We will assert that the correctness of the first two **if**-tests and their respective return values is self-evident, and focus on the remainder of the code. If control reaches the third **if**-test, the algorithm has established that there is some data that is buffered, but none of that data is currently deliverable. So at this point there are only two cases: either there *does* or *does not* exist any buffered object b that is also *waiting*.

Case 1: There *does* exist some such object b that is waiting. The definition of waiting tells us that object b can be delivered, and Theorem 6.7.6 gives us an algorithm for accomplishing this. Therefore, we should return **true**. ❑

Case 2: If there does not exist some such object b that is waiting then each of the buffered objects has as least one reliable predecessor. Thus declaring things lost would not help in any way, and we should return **false**; thus the **then-**clause of the third **if**-test is correct.

Theorem 6.7.8 The *isAnythingDeliverable*() operation can be implemented in constant time.

Proof of Theorem 6.7.8: The first **if**-test can be calculated in $O(1)$ time.

The second **if**-test can be calculated in $O(1)$ time as follows: the PO/PR receiver maintains a count of how many items are buffered. The counter is incremented when a TPDU arrives, and decremented when a TSDU is delivered.[73] If there are no buffered items at all, then the return value of *isAnythingDeliverable*() should be **false**.

The third **if**-test can be calculated in constant time, by simply checking whether the list of *waiting* objects is empty. Therefore the entire operation can be implemented in constant time. ❑

So far, we have shown that the pseudocode for the *isAnythingDeliverable*() operation for the basic PO/PR algorithm (Figure 6.15) is correct, and that this operation can be performed in constant time. We have also shown that the processing that was added to the *getNextTSDU*() operation that keeps track of the *numCOURPs* values can be amortized in such a way that we do not exceed our $O(n+e)$ upper bound for a period.

Proofs related to computing the $L(y)$ set via a DFS of the $TRPO^T$.

We now proceed to the other two operations. First Theorem 6.7.9 shows that the extra processing added to find the $L(y)$ set of a waiting item does not exceed $O(n+e)$ per period. Finally, we establish the correctness of the *getNextTSDU*() operation by showing that it will always deliver exactly one object per invocation.

Theorem 6.7.9: The total worst-case cost of computing DFS search trees from the $TRPO^T$ is bounded by $O(n+e)$, and can be amortized over the cost of the *getNextTSDU*() operations.

Proof of Theorem 6.7.9: It is sufficient to show that the adjacency list of each node in the $TRPO^T$ can never be traversed more than once, since this would limit the

---

[73] Note that TPDUs and TSDUs are in a one-to-one relationship in POCv2, in contrast with TCP, where this is not always the case.)

total processing to $O(n+e)$, a quantity that can be amortized over the *getNextTSDU*() operations. Note that the computation of the DFS search tree based on the $TRPO^T$ is invoked *only* when the *getNextTSDU*() operation is invoked at a time when some waiting element b must be delivered. When this is done, no further calculations of DFS search trees over the $TRPO^T$ can be made until after the queue *deliverOrDeclareLost* is empty, which ensures that all the elements of $L(b)$ have already been resolved (i.e., the predecessors of b, including b itself). Thus, if any future DFS of the $TRPO^T$ visits some element x in $L(b)$, element x is guaranteed to be already resolved, and its adjacency list will not be traversed. ❑

As a side-comment on Theorem 6.7.9, we note that unless there is a *getNextTSDU*() operation at a time when no data is deliverable, the computation of DFS trees over the $TRPO^T$ never takes place at all, so we are truly computing a worst-case running time.

Note that the object we end up delivering first may or may not be the one that we chose at the step “choose y = an arbitrary waiting item”. Suppose object x and object y are both objects with zero unresolved reliable predecessors, and $x \prec y$ in the PO. If y arrives before x, then y may be the arbitrary waiting item that is chosen. Object x will be part of the sorted $L(y)$ set, notwithstanding the fact that it has arrived. Since objects are then resolved in the order in which they were topologically sorted, object x will get delivered before object y, and the *getNextTSDU*() operation will return before object y is delivered. On a subsequent invocation of the *getNextTSDU*() operation, if deliverable objects have arrived in the meantime, they will have been placed on the output queue. Any objects still sitting in the *deliverOrDeclareLost* queue will either have been transferred to the output queue as they arrive, or will just stay on the *deliverOrDeclareLost* queue until the next time that it is necessary to begin resolving loseable items. In any case, as Theorem 6.7.10shows, some object *x* will always be delivered. This object *x* will be one that was *waiting* at the time when the set *L*(*y*) was computed. Before that object *x* is delivered, all of *x*'s predecessors will have been previously either delivered or declared lost.

Theorem 6.7.10: The operation *getNextTSDU*() will always deliver at least one object; that is, after the "wait" for *isAnythingDeliverable* to be true, either (1) there will be a deliverable object already on the output queue, or (2) there will be a waiting object *y*, and some object *x* will subsequently be delivered. As a consequence, the final "assert(**false**)" statement of the pseudocode in Figure 6.2 should never be reached.

Proof of Theorem 6.7.10: Theorem 6.7.6 already established the basic soundness of the approach of computing *L*(*y*) and resolving all elements in that list. All that is needed to establish the claims in this theorem is to fill in the details.

We wait until the *isAnythingDeliverable*() routine returns **true.** Inspection of this routine indicates that once it returns **true**, there will either be a buffered or a waiting item. If there is a buffered item, we deliver it, so the only part of the claim still in question is whether it is possible, when the code enters the "**while** (**true**)" loop, for the "assert(**false**)" statement to be reached before some object *x* can be delivered. Since all items on this list are unreliable, and can be declared lost if necessary, the only way that we can reach the "assert(**false**)" statement is if, when the loop is entered, no element on the *deliverOrDeclareLost* queue is a buffered object. If there were some buffered object *x*, on the *deliverOrDeclareLost* list, by Lemma 6.7.5, at the time that *x* was placed on the list, all elements of the set *L*(*x*) would have been placed on the list in a position preceding *x*. As a result, any unresolved predecessors of *x* would have been resolved before *x* reached the front of the list, and *x* would have been transferred to the output queue, and delivered.

However, by Theorem 6.7.6, the elements placed on this list constitute a topological sorting of all the predecessors of some buffered object, and then the buffered object itself. Therefore, the last element on this list is always guaranteed to be a buffered object at the time the initial list is constructed, and until that list is entirely resolved, no new elements can be placed on that list. In addition, by Theorem 6.7.6, the final element on that list must be the *y* element from which the list was constructed, and everything that precedes *y* on this list is a predecessor of *y*. The only way for *y* to leave the *deliverOrDeclareLost* list is if *y* were transferred to the *output* queue (which cannot happen until all of *y*'s predecessors have been resolved) or to reach the front of the list without being so transferred. Object *y* cannot reach the front of the list without being transferred to the output queue, since *y* is a buffered object, and if everything in front of *y* on the list has been resolved, *y* will have in-degree 0. Furthermore, anytime the *getNextTSDU*() operation is invoked, it is impossible for the *deliverOrDeclareLost* queue to contain any of the elements of *L*(*y*) unless object *y* is also sitting in the queue following these elements.

Therefore, we can never enter the "**while** (**true**)" loop under any circumstance except the one where at least one of *y*'s predecessors needs to be declared lost before *y* can be delivered. Under these circumstances, at least one object (not necessarily *y*) will always be delivered before the loop terminates.❑

6.7.3 Comparison/Contrast of PO/PR-DEL-BASIC with PO/R-RCV-RESEQ

It is instructive to compare and contrast the PO/PR-DEL-BASIC algorithm with the PO/R-RCV-RESEQ algorithm from which it was derived, both in terms of how the algorithm works, and in terms of running time—both actual and amortized worst cases for each operation.

Both algorithms keep track of when objects should be placed on the output queue by tracking when their in-degree changes and when they arrive. In addition, the algorithms for the PO/PR receiver also track when objects meet the definition of waiting by tracking the *numCOURPs* of each object along with each object's arrival. Both of these algorithms essentially constitute an execution of the DAGITS algorithm for topologically sorting the partial order: in the first case, we are actually resolving each object *x* by delivering or declaring *x* lost, and releasing x's successors. In the second case, we maintain a shadow of the original graph *G*, which we call G'. The reliable objects in G' are resolved at the same time they are resolved in *G*, while each unreliable object in G' is resolved as soon as it has no unresolved reliable predecessors.

The extra processing necessary to maintain the shadow graph G' actually gets scheduled at two times: in the *init*(*G*) routine (which is already $O(n+e)$, not amortized) and in the *releaseSuccessors*() procedure. For the PO/R-RCV-RESEQ algorithm, the *releaseSuccessors*(*x*) routine is $O(d)$, where *d* is the out-degree of *x* in *G* (i.e., the *TRPO*). Since each successor is released only once, we add the processing for each call to *releaseSuccessors*() together, resulting in a total of $O(n+e)$ for the entire processing of a single period.

For PO/PR-DEL-BASIC, a given *releaseSuccessors*() operation may require $O(n)$. As an example, consider again the PO consisting of an antichain of $n/2$ reliable objects followed by a chain of $n/2$ unreliable objects. Each *releaseSuccessors*() operation for the reliable objects (except for the last one) will require only constant time to decrement the *numCOURPs* of the first unreliable object, however the last invocation of *releaseSuccessors*() on a reliable object will take $O(n)$ time to traverse the chain of the $n/2$ remaining unreliable objects. However, note that this processing occurs only once over the course of the algorithm, and the total amount of processing is still $O(n+e)$. In practice, if multimedia documents are constructed with a mix of reliable and unreliable objects, and the total number of objects is kept modest ($n \leq 1000$), one would not expect an occasional $O(n)$ “hit” such as the one described here to be a serious concern.

6.7.4 PO/PR-DEL-BASIC: Section Summary

In this section, we have provided algorithms for the POCv2 receiver that implement partial order and partial reliability (with the POCv2 PR semantics). These algorithms accomplish this with a running time of $O(n+e)$ per period (amortized) by using:

- two parallel instances of the DAGITS algorithm, and
- an instance of DFS over the transpose of the transitively reduced precedence graph

As with PO/R-RCV-RESEQ, the *init*() operation requires time $O(n+e)$, and the *isAnythingDeliverable*() and *processIncomingTPDU*() operations run in constant time (without amortization). Over the course of the algorithm, the *getNextTSDU*() operation always invokes *releaseSuccessors*(x) for each object in the partial order; each call to *releaseSuccessors*(x) is $O(d)$ (not amortized), where d is the out-degree of x, and may additionally require $O(n)$ (not amortized) in the worst case for the updating of the *numCOURPs* fields of successor objects. However, this additional processing can be amortized to the previous $O(d)$ running time, or $O(n+e)$ over the entire period. Each *getNextTSDU*() operation may also invoke, if needed to declare predecessors of a waiting object lost, a DFS over the loss-candidates associated with some waiting object. This DFS processing can also be amortized over the $O(d)$ running time of the *releaseSuccessors*() operations.

Therefore, while in the case of PO/PR receiver processing, individual *releaseSuccessors*() or *getNextTSDU*() operations may require as much as $O(n)$ time, the total *amortized* running time of processing a period of n objects with e edges in the corresponding TRPO is unchanged as compared to the processing time required for the PO/R-RCV-RESEQ algorithm.

In the next section, we extend the PO/PR-DEL-BASIC algorithm to incorporate explicit release, and the stream abstraction.

6.8 PO/PR-DEL-FULL: Adding streams and explicit release to POCv2

POCv2 incorporates three features that, in isolation, are relatively simple to describe and implement, namely, (1) a particular semantics for partial reliability, (2) explicit release synchronization, and (3) a stream abstraction for describing the incorporation of larger objects into partial orders. However, the interaction of these features presents certain problems.

Overview of Section 6.8

Section 6.8.1 describes some of the questions that must be resolved in defining the interactions of these three features, and the definitions adopted for purposes of the current investigation. Section 6.8.2 describes the main difficulty with integrating the three features. We then describe two approaches to extending the algorithm of Section 6.7 to include these three features:.

- Section 6.8.3 describes an approach that *preserves the computation efficiency*, but requires *giving up two desirable properties* of the properties of the POCv2 PR semantics.
- Section 6.8.4 describes an approach that *preserves the full POCv2 PR semantics* but requires an *inefficient brute-force computation*. Determining whether an efficient algorithm exists for the full POCv2 PR semantics remains an open problem.

Section 6.8.5 concludes Section 6.8 with suggestions for future work, including possible directions to pursue in finding a more efficient algorithm for implementing the full POCv2 PR semantics.

6.8.1 Integrating POCv2's PR class with explicit release and stream objects

The following questions arise when we consider how to integrate the PR reliability class with explicit release synchronization and the stream abstraction.

Interaction of Reliability Classes with Stream Objects

The first question that arises is whether all cells of a stream object should be required to have the same reliability class. It may be useful to allow the various cells of a stream to have different reliability classes. For example, in an MPEG video stream, it may be useful to send the "I" frames (complete images) with higher reliability than the "P" and "B" frames that represent deltas from some neighboring "I" frame. However, for purposes of simplicity, in the current work we answer “yes”, requiring that all cells of a particular stream object carry the same reliability class, and defer the complication of multiple classes within a stream object to future work.

Another question that arises is whether POCv2 should enforce an “all or nothing” semantics for PR stream objects. That is, once a stream object with reliability class PR has begun to be delivered, should it then be treated as reliable? We clearly cannot implement such a semantics for stream objects with reliability class U, since there is no mechanism for retransmitting lost cells of a stream object with class U. Therefore we allow each cell to be delivered or declared lost independent of all other cells, although we will require that the individual cells that comprise a particular stream object be resolved (delivered or declared lost) in linear order.

Integration of stream objects with explicit release

In the context of explicit release, an object is not resolved until it has been both delivered *and* its successors have been released. To clarify this further, consider the following example:

> Suppose that explicit release synchronization is being used to synchronize the end of an audio object, *x*, where all of *x*’s cells are of class PR. The first half of the cells for object *x* have been given to the audio device, but have not yet finished playing. An object *y* follows object *x* in the partial order and is now *waiting;* the remaining cells of object *x* seem to have been lost or delayed, and the application is now requesting data.

The question is: should the transport layer declare the remainder of the cells of *x* lost and deliver y, or should the transport layer wait for a signal from the application that the first half of the cells have been resolved? If we proceed to declare the second half of the cells of *x* lost and release the successors of x, including y, then we may deliver *y* before the cells of *x* that *were* delivered have finished playing. This violates the synchronization semantics of the document, and cannot be allowed.

Therefore, there must be a requirement that within an object, any cells that were *delivered* must be explicitly released before successor cells of the same object can be declared lost. If implemented naively as an explicit release operation per cell, this could be rather costly in terms of increasing the interaction between the application and the transport layer, but fortunately, there is an efficient solution as we now explain.

Streaming vs. Stalled objects, and the underflow notification

Consider the case of an audio stream where each cell contains 20ms of audio. Requiring the application to explicitly release each cell might require 50 extra operations across the TSAP every second, effectively doubling the number of TSAP operations required for an audio stream.

Instead, we impose a lesser burden on the application that desires synchronization of U or PR stream objects; the application must notify the transport layer anytime the playout of the stream object *underflows*. For example, in the case of an audio object, the object underflows if the queue of data flowing to the audio device empties out. With this extra information from the application, we can now make a determination as to whether a stream object is *streaming* (i.e. currently the process of playing out content) or when it is *stalled* (a stream object that is ready to play, but is waiting for cells to arrive.)[74] The synchronization relationships can then be preserved with the following rule:

---

[74] Formal definitions of *streaming* and *stalled* appear in Section 6.8.3

- unreliable cells or objects that have predecessors in the streaming state may not be declared lost,
- unreliable or partially reliable cells from a stalled object can be declared lost.

Essentially, an underflow notification serves as an explicit release for all outstanding cells. The underflow notification provides the transport layer with sufficient information to preserve the synchronization relationships, because it is only at underflow points that an explicit release is needed at a finer granularity than that of complete objects in the partial order. A cell-level explicit release serves only to determine when to declare successor cells within a stream object lost. Until an underflow actually occurs, there is still the hope that the next undelivered cell in the stream may yet show up and be delivered, preventing—or at least, postponing—an impending underflow.

What running time can be achieved when incorporating all three features?

First, we should recognize that when incorporating the stream abstraction, it is necessary to add the number of cells, c, to our notation. Therefore, instead of seeking an $O(n+e)$ running time, we may seek an $O(n+e+\mathrm{c})$ running time. Whether or not this goal is achievable is currently an open problem. In this section, we will describe both the running time that is currently acheivable, as well as the barriers to reaching the goal of $O(n+e+c)$.

First, we note that re-sequencing the cells within a stream object is equivalent to the problem of reordering TCP segments. For reasons that were explained earlier in Section 6.1.3, a simple linear search of the out-of-sequence packets is the preferred technique to solve this problem; clearly this technique cannot guarantee an $O(n+e+c)$ running time.

However, since our focus is the analysis of the processing that pertains directly to the implementation of partial reliability, partial order and explicit release, we will account for the resequencing of cells within a stream object separately from the remainder of the processing; if we do this, $O(n+e+c)$ is at least a conceivable goal.

6.8.2 The main difficulty: explicit release of streaming objects

The problem that arises is with streaming objects. The correctness of the O(n+e) algorithm presented in Section 6.7 depends on the following property: if an object is determined to be waiting, meaning that all of its unresolved predecessors are unreliable, then if all of the unresolved predecessors of that object are visited in the order of a topological sort (determined by a DFS of the $TRPO^T$), then some object will become deliverable. However, when we admit the possibility that one or more of these unresolved predecessors may include a streaming object, then this property is violated. The next two sections describe two options to resolve this problem.

The first option, described in Section 6.8.3 is to relax two requirements of the POCv2 PR semantics, namely

(1) that an object should never be declared lost unless such declaration will result in the immediate delivery of some data

(2) that if an object is waiting and has no unconstrained predecessors, then it will always be deliverable.

We considered this approach undesirable, however it has the advantage that when this approach is taken, the running time for the algorithms presented in Section 6.7 is preserved.

The second option, described in Section 6.8.3, is to maintain all of the requirements of the POCv2 PR semantics, regardless of the computation time required. Implementing this option efficiently is currently an open problem.

6.8.3 An efficient algorithm that sacrifices two desirable properties

In this section, we sketch an algorithm called PO/PR-DEL-OPTION1, which is a modification of the PO/PR-DEL-BASIC algorithm from Section 6.7. In this algorithm, we add linear-time resequencing of cells within stream objects (as in the standard practice for TCP), and additional constant time operations per protocol operation. Therefore, the algorithm can be considered efficient. However, we also sacrifice two desirable properties of the POCv2 PR reliability semantics. In this section, we first sketch the algorithm, and then explain two properties of the definition of the POCv2 PR class that are violated.

Sketch of algorithm PO/PR-DEL-OPTION1

To develop Algorithm PO/PR-DEL-OPTION1, we start with PO/PR-DEL-BASIC algorithm from Section 6.7. Certain modifications are necessary to implement the stream abstraction, and to provide the necessary bookkeeping so that the current stream state of each object can be determined in constant time. These modifications are straightforward, and they add little insight; we therefore omit them.[75] Instead, we focus on three specific modifications that illustrate the problems

---

[75] The actual code for the POCv2 implementation used in the ReMDoR experiments contains the modifications for explicit release and the stream abstraction (but not the PR semantics) and is available on-line. Therefore, we concentrate only on the PR semantics in this section.

involved in modifying the PO/PR-DEL-BASIC algorithm to provide the features necessary for the full POCv2 specification.

The first modification is to add an operation *reportUnderflow*(*objNum*) that supports explicit release synchronization for unreliable and partially reliable stream objects. Let *x* be the object referred to by the integer objNum. The application may—and in some cases, must—call this operation any time that object *x* is starving for additional cells; that is, all cells that the application has read from object *x* have been completely presented. This operation signals the POCv2 receiver that no synchronization relationships would be violated by declaring lost the next cell (or chain of consecutive cells) from this stream object.

Whether the application *may* invoke the *reportUnderflow*(*objNum*) operation or *must* invoke this operation depends on the reliability class of the object *x* referred to by *objNum*:

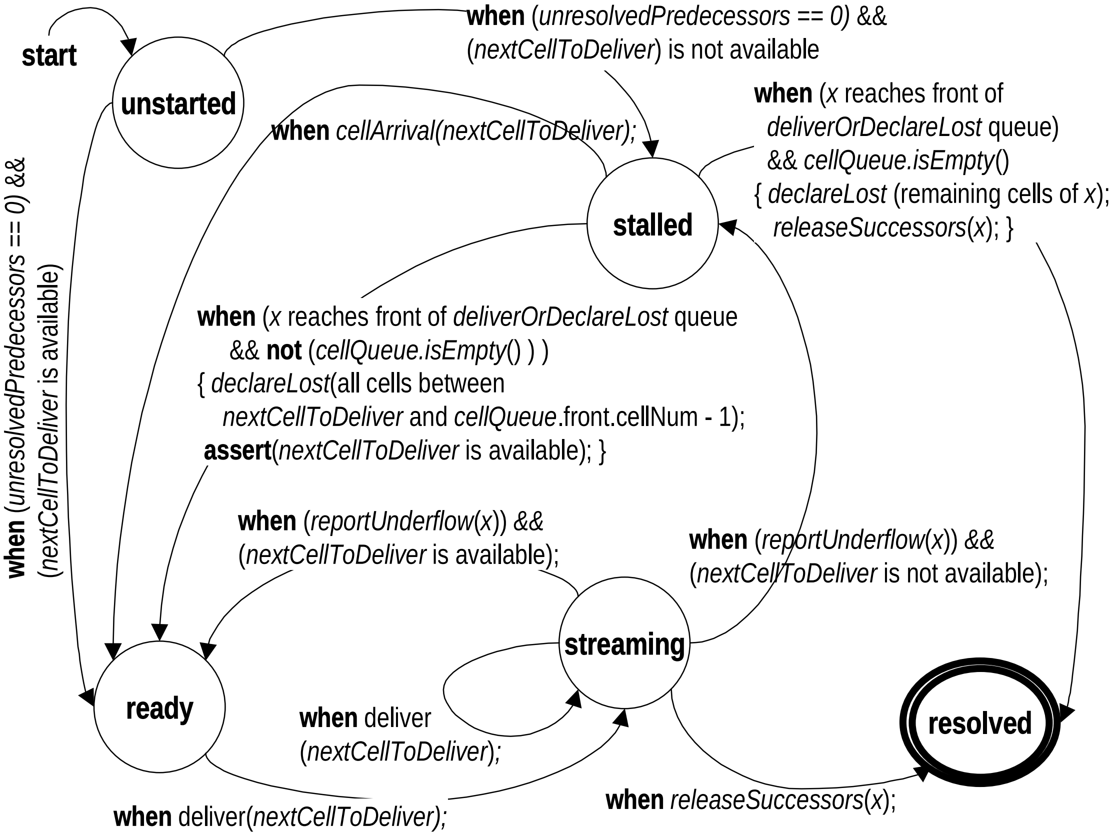
- If object *x* is unreliable, then invocation of this operation is necessary anytime object *x* contains more than one cell, and the network is lossy. This invocation is necessary because unreliable objects are not retransmitted, and if a cell other than the first cell is lost, without a call to this operation, the transport protocol will deadlock waiting for an object that will never arrive.

- If object *x* is partially-reliable, than this operation is optional; calling it allows the PO receiver more flexibility in trading off reliability for delay.

- If object *x* is reliable, than this operation is meaningless, and is ignored

The next two modifications take place in the implementation of the *getNextTSDU*() operation. We require the following definition:

Definition 6.8.1: The function *currentObjectState*(*x*) maps each object x in the current period at the PO Receiver to one of the following *stream states*:{*unstarted*, *streaming*, *stalled*, *ready*, *resolved*}. The stream state of each object *x* is defined by the finite state automata shown in Figure 6.22.



start
unstarted
when (*unresolvedPredecessors == 0)* &&
(*nextCellToDeliver*) is not available
when *cellArrival(nextCellToDeliver);*
when (*x* reaches front of
deliverOrDeclareLost queue)
&& *cellQueue.isEmpty*()
{ *declareLost* (remaining cells of *x*);
releaseSuccessors(*x*); }
stalled
when (*unresolvedPredecessors == 0)* &&
(*nextCellToDeliver* is available)
when (*x* reaches front of *deliverOrDeclareLost* queue
&& **not** (*cellQueue.isEmpty*()))
{ *declareLost*(all cells between
nextCellToDeliver and *cellQueue*.front.cellNum - 1);
assert(*nextCellToDeliver* is available); }
when (*reportUnderflow*(*x*)) *&&*
(*nextCellToDeliver* is available);
when (*reportUnderflow*(*x*)) *&&*
(*nextCellToDeliver* is not available);
streaming
ready
when deliver
(*nextCellToDeliver*);
resolved
when *releaseSuccessors*(*x*);
when deliver(*nextCellToDeliver);*

Figure 6.22 Finite State Automata for stream states

For sake of space, we make two claims about the Finite State Automata in Figure 6.22 without formal argument. Claim 1: We can add bookkeeping operations to the pseudocode for the algorithm in Section 6.7 to keep track of the current stream state of each object. Claim 2: This extra bookkeeping does not increase the asymptotic running time per operation beyond the analysis presented in Section 6.7.

We can now describe the remaining two modifications, both of which take place in the *getNextTSDU*() operation as shown in Figure 6.23. The bullets in this figure indicate the lines that have been modified as compared to the previous version from Figure 6.2.   To implement explicit release, we remove the line of code that implicitly released the succesors of each object after it was delivered, and instead make the *releaseSuccessors*() and *reportUnderflow*() operations directly available to the transport service user (the receiving application).

Finally, we must prevent cells from being delivered or declared lost whenever predecessor cells from the same object are currently streaming.   Recall from Section 6.7 that the *deliverOrDeclareLost* queue contains a list of topologically sorted objects, with the property that one or more of the objects in the list (at least, the final object) has deliverable data.   The second section of added code in Figure 6.23 specifies that if an object comes to the front of this queue while it is streaming, then the PO receiver must suspend declaring objects lost until that object is no longer streaming.

This latter modification prevents synchronization violations that would result if streaming objects could be declared lost while still streaming. Furthermore, this modification will not cause a deadlock provided that either the application eventually either reports an underflow, or releases the successors of the object; nor will it prevent or unnecessarily delay the delivery of any objects that do arrive. However, this algorithm fails to implement two desirable properties of an algorithm for the POCv2 PR class, as we explain below.

```
operation getNextTSDU() returns TSDU:

…
[This section of code exactly as in Figure 6.2]
…

          while (true)
          {
              if (not output.isEmpty()) // if a waiting object become deliverable
              {
                  x = output.remove;
●                 // Note: do not release successors here;
                  // wait for explicit release
                  return encapsulated TSDU from inside x
              }
              else
              {
●                 x = deliverOrDeclareLost.peek();
●                 if (x is streaming)
●                     return null;
                  x = deliverOrDeclareLost.remove();
                  declareLost(x);
                  releaseSuccessors(x.objnum);
              }
          }
 assert(false); // by theorem 6.7.9, we should never reach this statement
```

Figure 6.23 Modified psuedocode for *getNextTSDU()*, option 1

The • symbol indicates changes from the psuedocode presented in Section 6.7

The first property we sacrifice: no premature loss declarations

PO/PR-DEL-OPTION1 violates the requirement that no cell of any object will be declared lost unless and until this action directly results in the delivery of some other cell. As an example, consider the following scenario: unreliable objects *x* and *y* both precede z, which has data available. Object *x* is a one-cell object, while *y* is a multiple-cell object that is currently streaming. The DFS of the TRPOT that locates the predecessors of *z* places *x* on the *deliverOrDeclareLost* queue in an earlier position than *y*. As a result, *x* is declared lost before *y* is determined to be streaming. Object *x* has now been “sacrificed” earlier than was necessary. It is possible that *x* could arrive before *y* is finished streaming; in this case, *x* would have been able to be delivered if only the PO/PR receiver had not been hasty to declare it lost. It is undesirable for an algorithm providing PR delivery to declare an object lost unless that declaration results in a specific tangible benefit, viz., a reduction in delay for some other specific object.

The second property we sacrifice: no deliverable data waiting for unreliable data

The second property we sacrifice is the one that the delivery of data should never be delayed by unreliable predecessor data. This property is violated in the modified algorithm. Given that the delivery of TSDUs is prevented any time the head object of the *deliverOrDeclareLost* queue is streaming, the delivery of a TSDU by the *getNextTSDU*() operation can no longer be guaranteed whenever there is waiting data.

To see how the property is violated, first review how the property is shown to be maintained by the PO/PR-DEL-BASIC algorithm. Theorems 6.7.7 and 6.7.10 proved the correctness of the *isAnythingDeliverable*() and *getNextTSDU*() operations, respectively. The argument is that if there is any waiting data, *isAnythingDeliverable*() will return **true**, and if the operation *getNextTSDU*() is invoked at a time when *isAnythingDeliverable*() would return **true**, at least one cell will be delivered. The operation *getNextTSDU*() chooses an arbitrary waiting item *y* as the basis for the DFS of the $TRPO^T$ that fills the *deliverOrDeclareLost* queue.

In the modified algorithm, this arbitrary choice is invalid. Suppose there are several waiting objects (here we assume that the definition of “waiting” is not modified to exclude objects with streaming predecessors.) Partition the waiting objects according to whether or not each object has a streaming predecessor. Choosing any of the objects that lacks a streaming predecessor results in the delivery of some cell by the *getNextTSDU()* operation. However, choosing any of the other objects may or may not result in delivery of a cell. If some predecessor of the chosen object has deliverable data that ends up in the *deliverOrDeclareLost* queue before any and all streaming predecessors of the chosen object, then a cell is delivered. If not, then the property of “no deliverable data waiting for unreliable data” has been violated: a waiting object that could have been delivered exists, yet the *getNextTSDU*() operation returns with no cell.

6.8.4 A brute-force algorithm that implements the full POCv2 PR semantics

To develop an algorithm with the desired properties, we face several challenges.

- First, we must ensure that the *isAnythingDeliverable*() operation can distinguish between a waiting object that *has* a streaming predecessor, and one that does not.
- Second, we need to ensure that the *getNextTSDU*() operation returns only the latter kind of object as the basis for the DFS over the $TRPO^T$.
- Third, and perhaps most challenging, we must deal with the fact that the stream state of an object is dynamic, and can therefore change between invocations of *getNextTSDU*().

To deal with this third challenge, we would like to ensure that the *deliverOrDeclareLost* queue is completely emptied after each call to *getNextTSDU*(), so that we can treat the stream state as static when reasoning about the contents of this queue. To ensure that the queue is emptied, we require that no waiting object with a waiting predecessor should be chosen as the basis of the DFS over the $TRPO^T$— instead, the waiting predecessor should be chosen. If the chosen waiting object *y* has no waiting predecessor and no streaming predecessors, then the final loop in *getNextTSDU*() will always have the same outcome: a sequence of unreliable/partially reliable objects is declared lost, and then a cell from the waiting object *y* is delivered. With this extra requirement, *deliverOrDeclareLost* becomes a true queue since it will be filled and emptied in strict FIFO order within a single call to *getNextTSDU*().

Sketch of algorithm PO/PR-DEL-OPTION2

We now sketch PO/PR-DEL-OPTION2, a modification of PO/PR-DEL-BASIC that preserves all of the properties of the proposed POCv2 semantics for the PR reliability class, but at the expense of an inefficient running time. We do not propose PO/PR-DEL-OPTION2 as an algorithm to be implemented. Rather, the purpose of PO/PR-DEL-OPTION2 is to provide a starting point for future work towards a more efficient algorithm, or towards lower-bound proofs showing that more efficient algorithms cannot be found.

As with our sketch of PO/PR-DEL-OPTION1, we omit certain housekeeping details, and focus only on the changes that are crucial to the running time, and key properties of POCv2's PR semantics. We add just one new local procedure, as shown in Figure 6.24. We then place calls to this new procedure into *isAnythingDeliverable*() and *getNextTSDU*() as shown in Figures 6.25 and 6.26.

The difficulty now is to find an efficient implementation of the procedure *findWaitingObjectWithNoWaitingOrStreamingProperPreds*(). We consider this problem in the next section.

```
local procedure findWaitingObjectWithNoWaitingOrStreamingProperPreds()
returns pointer to object;
            // returns pointer to object meeting the necessary criteria, or
            // null if no such object exists.
            // Used in both isAnythingDelierable()and getNextTSDU() to choose object
            // that serves as basis of the DFS on $TRPO^T$
```

Figure 6.24 Procedure *findWaitingObjectWithNoWaitingOrStreamingProperPreds*()

```
operation isAnythingDeliverable() returns boolean:
{
    if (not output.empty() or count == n) return (true);
    else if (there are no buffered items)
            {return false} // there's no data at all
    else if (there exist no waiting items) // see defn 6.6.2; thm 6.7.1
            {return false} // there's data, but each object is waiting on a least one
            // reliable predecessor, so declaring things lost won't help
    else {
●               y = findWaitingObjectWithNoWaitingOrStreamingProperPreds();
●               if (y == null)
●                   return false;
●               else
●                   return true;        // there is at least one item we could deliver
            if we
●                                       // declared its predecessors lost
             }
}
```

Figure 6.25 Modified pseudocode for *isAnythingDeliverable()*, option 2

The • symbol indicates changes from the psuedocode presented in Section 6.7

```
operation getNextTSDU() returns TSDU:
local variable TPDU pointer tpdu;
if (count == n) return nil; // all objects have been delivered
wait(isAnythingDeliverable);
            // if false, sleep; recheck condition after each processIncomingTPDU() call;
```

```
                    // in practice, the sleep can be avoided by never calling getNextTSDU()
                    // without first checking isAnythingDeliverable().
    if (not output.isEmpty())
        {
                    tpdu = output.dequeue;
●                   // Note: do not release successors here; wait for explicit release
                    return encapsulated TSDU from inside tpdu
        }
    else // deliver a waiting item (after possibly declaring some items lost)
        {
        if (deliverOrDeclareLost.isEmpty())
        {
●                   y = findWaitingObjectWithNoWaitingOrStreamingProperPreds();
●                   assert(y != null); // implied since we waited for isAnythingDeliverable();
                    fillDeliverOrDeclareLostQueueWithSortedLSet(y);
                    // find the set L(y), consisting of all of y's undelivered unreliable
                    // predecessors, and topologically sort them according to the
                    // partial order, and place these items on the
                    // deliverOrDeclareLost queue.
        }
        while (true)
        {
                    if (not output.isEmpty()) // if a waiting object become deliverable
                    {
                        x = output.remove;
●                       // Note: do not release successors here; wait for explicit release
●                       assert(x == y); // if no object predecessor of y was waiting, first object
●                                       // to become deliverable will be y
                        return encapsulated TSDU from inside x
                    }
                    else
                    {
                        x = deliverOrDeclareLost.remove();
                        declareLost(x);
                        releaseSuccessors(x.objnum);
                    }
        }
●assert(false); // we should never reach this statement
```

Figure 6.26 Modified pseudocode for *getNextTSDU()*, option 2

The • symbol indicates changes from the pseudocode presented in Section 6.7

Running time of algorithm PO/PR-DEL-OPTION2

The key to the running time of PO/PR-DEL-OPTION2 is to implement the procedure *findWaitingObjectWithNoWaitingOrStreamingProperPreds*() efficiently. The corresponding problem in the PO/PR-DEL-BASIC algorithm was to find *any* waiting object; this could be done in $O(1)$ time by simply keeping a list of such objects, and returning the front element of the list. The time necessary to maintain the list was amortized to other operations. However, in PO/PR-DEL-OPTION2 the problem is more difficult.

First, let us suppose that in addition to the adjacency list representation of the *TRPO*, the PO receiver also initializes a transitively closed adjacency matrix representation denoted by *TCPO*. The benefit of initializing the *TCPO* for each epoch is that it subsequently permits the PO receiver to determine in $O(1)$ time whether $x \prec y$ for arbitrary objects x and y. The cost of this initialization is either

- an additional $O(n^2 \log n)$ bits in the transmission of the service profile, plus $O(n^2)$ initialization cost, or else
- an $O(n^3)$ computation to compute the *TCPO* directly from the adjacency list representation of the *TRPO* sent in the service profile.

Now, consider the problem of finding a waiting item that has no streaming predecessors or waiting predecessors. Given the availability of *TCPO*, the brute force approach to either finding such an item, or determining that no such item exists is shown in Figure 6.27. As can be seen from the nested loop structure, this implementation of *findWaitingObjectWithNoWaitingOrStreamingProperPreds* has a worst case running time of $O(w^2 + sw)$. Recall that our goal is to be able to amortize the time required by each operation to $O(1)$ per operation. Instead, since both s and w could be $O(n)$, we have a worst case running time that could be as much as $O(n^2)$.

Furthermore, there seems little hope that this time can be amortized: because the stream states of object can change between invocations of *isAnythingDeliverable*() and *getNextTSDU*(), it seems unlikely that an incremental data structure can be built to avoid repeating the expensive operations. Therefore, we are less than satisfied with this running time, a fact which leads us to suggest three specific lines of future investigation, as described in the next section.

```
local procedure findWaitingObjectWithNoWaitingOrStreamingProperPreds()
returns pointer to object;
    // returns pointer to object meeting the necessary criteria, or
    // null if not such object exists.
    // Used in both isAnythingDelierable()and getNextTSDU() to choose object
    // that serves as basis of the DFS on TRPOT
{
    foreach (w in the list of waiting items)
    {
        boolean noWaitingOrStreamingPreds = true;
        foreach (s in the list of streaming items) while (noWaitingOrStreamingPreds)
                if ($s \prec w$)
                    noWaitingPreds = false;
        foreach (x in the list of waiting items) while (noWaitingOrStreamingPreds)
                if ($x \prec w$)
                    noWaitingOrStreamingPreds = false;
        if (noWaitingOrStreamingPreds)
                return w

    }
    return null;
}
```

Figure 6.27 Brute-Force approach to PO/PR-DEL-OPTION2

6.8.5 Future work related to the POCv2 PR reliability class

There are several unresolved questions related to the POCv2 PR reliability class. The overarching question is whether the proposed POCv2 definition of the PR reliability class is useful—that is, can the POCv2 definition of PR offer performance benefits that are perceivable by an end-user? This question is best resolved by experiments with PO/PR service similar to the ones for PO/R service presented in Chapter 5. The fact that we do not currently have an efficient algorithm to efficiently implement the POCv2 PR semantics in their full specification leads to three subproblems:

(1) **Is the algorithm in PO/PR-DEL-OPTION1, in fact, good enough?** It may be the case that the problems foreseen with PO/PR-DEL-OPTION1 rarely arise in practice, or are mitigated by performance gains. This case can be investigated by implementing the PO/PR-DEL-OPTION1 algorithm in the UTL/ReMDoR framework, and conducting performance experiments.

(2) **Is there a better algorithm for PO/PR-DEL-OPTION2?** Several potential approaches for improving the efficiency of this algorithm are outlined below.

(3) **What do simulation results tell us about the overarching question, as to whether PO/PR can provide benefits perceptible by an end user?** We can use discrete event simulation to measure the expected performance of the ReMDoR application using PO/PR-DEL-OPTION2, assuming that the processing time is negligible. The outcome of such a simulation would tell us whether an efficient algorithm for PO/PR-DEL-OPTION2 would be a matter of practical interest or merely theoretical interest.

Approaches to improving the running time of PO/PR-DEL-OPTION2

There are several approaches to improving the performance of PO/PR-DEL-OPTION1. First, note that the algorithm does not need to check *all* elements of the *waiting* set, but only those that are minimal within the waiting set w.r.t. to the partial order. By analogy with a min-heap (which returns the minimum element of a totally ordered set in constant time, and allows inserts and deletes in time $O(\log n)$), we might pursue the implementation of a partially-ordered min-heap. This data structure would provide an operation to iterate through the list of the minimal elements in time $O(m)$, where m is the number of minimal elements currently in the set. It is clear that a circular list of multiple min-heaps could provide this operation; what is less clear is how efficiently inserts and deletes could be performed.

Suppose, however that such an operation could be implemented. Now the procedure *findWaitingObjectWithNoWaitingOrStreamingProperPreds*() can be sketched as shown in Figure 6.28. The algorithm may be able to take advantage of the fact that the list of minimal waiting objects, ℓ_1, and the list of streaming objects, ℓ_2 possess certain known properties: (1) they are both antichains, (2) they are both disjoint, and (3) no element of ℓ_2 precedes any element of ℓ_1. The fact that both lists are antichains limits the size of each list to the maximum width of the partial order, which is already an improvement over the previous algorithm (discounting the as yet unknown cost of implementing a partially-ordered heap.)

Future work may consider whether use of a *k*-dimensional representation of the partial order (representing the order as an intersection of *k* chains) can improve the running time of PO/PR-DEL-OPTION2. Such a representation may allow an efficient algorithm for *k*-dimensional partial orders. (Most partial orders have dimension 1 or 2, and partial orders with dimension 5 or more are extremely rare.)

```
local procedure findWaitingObjectWithNoWaitingOrStreamingProperPreds()
    returns pointer to object;
{
    let list $\ell_1$ = the list of streaming objects
    let list $\ell_2$ = the minima of the waiting objects.
    // now the following properties hold:
    //    $\ell$1  and $\ell$2 are both antichains
    //    $\ell$1  and $\ell$2 are disjoint (defn's of streaming and waiting are mutually exclusive)
    //    no element of  $\ell$2 precedes any element of list $\ell_1$ .
    find an element of $\ell_2$ with no predecessor in $\ell_1$, and return it
                OR determine that no such element exists and return NULL
}
```

Figure 6.28 Alternate pseudocode for PO/PR-DEL-OPTION2

6.9 Representation of partial orders (encodings, data structures)

One objection that can be raised to a PO/PR transport protocol is that there is overhead associated with the transmission of the service profile. To make efficient use of the bandwidth, it is desirable to represent this data structure with as few bits as possible. On the other hand, it is sometimes desirable to tradeoff efficient processing at sender and receiver for a larger number of bits in the protocol header (Chandranmenon and Varghese, 1995). In this section, we survey some of the techniques that can be used to represent these data structures, and the tradeoffs associated with these data structures. We then argue that the transitively-reduced adjacency list is the preferred representation for PO/PR transport protocol specification and implementation.

The question of how to represent a partial order *P* arises in at least three contexts:

- *Transmission:* how the partial order is represented in the PDUs exchanged between peer transport-layer entities, i.e. at connection establishment time.

- *Processing:* how the transport-layer sender and receiver represent the partial order internally during the data transfer phase.

- *Application Interface:* how the application represents the partial order when it is passed to the transport protocol through the API when the connection is requested.

Previous work on POC dealt only with the transmission of the PO and the sender and receiver processing; the API was not defined. ((Amer et al., 1993, 1994), (Marasli et al., 1996a: 1996b, 1997a, 1997b, 1998), and RFC1693)). These efforts used the transitively closed matrix representation for both transmission of the partial order during connection establishment, and for the internal representation of the partial order.

Table 6.2 lists various techniques that can be used to encode a partial order *P* over *n* objects, along with the number of bits required.

Table 6.2 A partial listing of POCv2 service primitives

Representation	Number of bits required	Class of partial orders for which this is valid	Reference
Full 0-1 adjacency matrix	n^2	any partial order	(well-known)
Upper Triangular 0-1 adjacency matrix	$n(n-1)/2$	any partial order where (1 2 3…*n*) is a valid linear extension	(Amer et al., 1993)
Two Total Orders (take intersection)	$2n\lceil \log_2 n \rceil$	Any series-parallel partial order	(Valdes, Tarjan and Lawler, 1982)
One Total Order $\cap$ with $\{1 \prec 2 \prec \ldots \prec n\}$	$n\lceil \log_2 n \rceil$	any partial order where (1 2 3…*n*) is a valid linear extension	easy extension of (Valdes, Tarjan and Lawler, 1982)
Adjacency List of Transitively-reduced precedence graph	$(n+e)\log\lceil n+1 \rceil$	any partial order	(well-known)

Two of the representations in this table require that 1..n be a valid linear extension of the partial order. On the one hand, this may not seem like a burdensome requirement, since the objects can simply be renumbered. However, for some applications, it may create a real inconvenience. Consider the screen-refresh example in Section 2.2.1. The user of the window system is free at any time to change the orientation of the windows in such a way that $\langle 1;2;3;4 \rangle$ is no longer a valid linear extension. To maintain $\langle 1;2;3;4 \rangle$ as a valid linear extension, both sending and receiving application would have to communicate a new mapping between the applications numbering scheme and the one used by the PO/PR protocol. This requirement would likely wipe out any gain realized by using an object representation that relies on the assumption of $\langle 0;1;2;\ldots;n-1 \rangle$ as a valid linear extension.

The adjacency list encoding is simply the concatenation of the adjacency lists for objects 0,1,…,*n-1* with each list preceded by its length. In the case of ReMDoR, the overhead of transmitting the partial order is negligible when put into context. Let *p* be the number of periods during which a given partial order *P* is in effect. The overhead for transmitting the partial order for the (n+e)(log n) representation is

$$\left(\frac{(n+e)\log_2 n}{np}\right)\text{bits} = \left(\tfrac{1}{p}\left(1+\tfrac{e}{n}\right)\log_2 n\right)\text{bits}$$

when amortized. While there is no corpus of multimedia documents that can be consulted to verify this, a reasonable guess at the value of e/n suggests that somewhere around 5 might be a reasonable upper bound, and that around 1000 is a reasonable upper bound for n. Thus, even if p is 1, we are looking at PO representations in the range of 60 bits, or around 8 bytes per object

Finally, we must consider how quickly we can convert one representation to another; Table 6.3 summarizes applicable results. $M(n)$ denotes the running time of $n \times n$ matrix multiplication. Currently, the best known time for $M(n)=O(n^{2.36})$(Cormen et al., 1990). A more practical algorithm for transitive closure due to Warshall takes time $O(n^3)$ (Cormen et al., 1990) and can be used to implement transitive reduction by applying techniques from (Aho, Garey and Ullman, 1972). To put this conversion in perspective, Table 6.4 shows some figures for the running time of this algorithm in practice, from an implementation that is hardly optimized (i.e., it contains a considerable amount of debugging code.) Even with this relatively unoptimized code, we can:

- process POs with n<= 100 with sub-second response time
- process POs with n<= approx. 200 elements in a couple of seconds (what would be considered a “fast” compile time for a programmer)
- process POs with n<= approx. 400 elements in around 10-15 seconds (which might be considered a “moderate” compile time for a programmer)

- process POs with n<= approx. 800 in a few minutes; say, the time it currently takes an average PC to complete a full virus scan on all the files of a reasonably large hard drive.

However, as *n* goes beyond 400, the running times moves into the realm of minutes—a user might need to get up and get a cup of coffee, check his/her email, or run a quick errand while the document is being compiled. As *n* moves to our hypothetical upper bound of 1000 elements, the running times are feasible, but unattractive (somewhere in the realm of 1 to 15 minutes), and most likely impractical beyond 1000.

While using a more sophisticated algorithm based on reducing transitive reduction and closure to matrix multiplication and applying something like Strassen's algorithm might help, a more practical solution might be to use the POCv2 concepts of "epochs" and "periods". The strategy would be to decompose the document into multiple partial orders, where the size of each partial order is determined by the following tradeoff:

- keeping *n* below some number that, in practice, allows the computation of transitive reduction and transitive closure within a reasonable running time (say, no more than 3 seconds)
- within that constraint, making the size of each period as large as possible to avoid the performance penalty imposed by the strict sequencing requirement between periods.

Once can envision a document authoring tool that allows the author to control this tradeoff in a manner similar to that used by compilers that can turn code optimizations on or off with a run-time option. Just as programmers often will turn off code optimization during the code/compile/test sequence during development, a document author might choose to decompose documents into smaller periods for faster compilation during the authoring process. When the document is *resolved* and ready to be placed on the server, the author could then select a higher target maximum for document decomposition, and run the necessary calculation off-line (overnight, if necessary) to provide better performance during document delivery.

In the end, we have chosen the transitively reduced adjacency list representation chiefly because it is a convenient representation for the algorithms at both sender and receiver, and the resulting PDU sizes are acceptable.

Table 6.3 Algorithms to convert between PO representations

We can convert this representation…	**…to this one…**	**…in time:**	**Reference for algorithm**
Adjacency matrix	Adjacency list	$O(n^2)$	well-known
Two Total Orders	Adjacency list (provided *P* is series-parallel)	$O(n+e)$	(Valdes, Tarjan and Lawler, 1982)
Matrix (or adj. list) with arbitrary transitivity	Transitively-reduced matrix (or adj. list)	$M(n)=O(n^{2.36})$	(Aho, Garey and Ullman, 1972)
Matrix (or adj. list) with arbitrary transitivity	Transitively-closed matrix (or adj. list)	$M(n)=O(n^{2.36})$	(Aho, Garey and Ullman, 1972)

Table 6.4 Time to compute transitive closure and transitive reduction in the ReMDoR parser for various values of *n* on Sun Ultra 10.

m (see note*)	**n (# of elements)**	**Time to compute transitive reduction**	**Time to compute transitive closure**
1	8	<1ms	<1ms
2	11	<1ms	<1ms
4	17	<100ms	<100ms
8	29	<100ms	<100ms
16	53	<100ms	<100ms
32	101	424ms	504ms
64	197	3.0sec	2.7sec
128	389	13.5sec	10.4sec
256	773	106sec	79sec
512	1541	14min 1 sec	10min 13sec
1024	3077	1 hour 53 min	1 hour 21 min
2048	6149	15 hours 9 min	11 hours 27 min

*The documents used for this experiment were generated by a script and were authored solely for the purpose of testing the system with large documents. They have the following storyboard: they present the numbers 1 through m with a one second pause between each number. After each number is drawn, it is erased before the next number is displayed, and the word "DONE" is displayed at the end. The number of elements in each such document is $n = 3m+5$.

6.10 Chapter summary and suggestions for future work

This chapter motivated, described, and provided proofs of correctness and running time analysis for several algorithms required for provision of a PO/R transport service. In particular, we showed how a particular view of the Topological Sort problem as a incremental process provides a foundation for several algorithms related to the implementation of partial order transport.

The chapter also described algorithms needed to integrate a PO/R transport service within the ReMDoR multimedia document retrieval system, including the algorithm for explicit release synchronization.

Several problems related to the implementation of the POCv2 PR semantics were described. As a basis of future investigation, we provided two algorithms: one that is efficient—$O(n+e+c)$ per period, amortized—but does not implement the full semantics of the POCv2 PR reliability class, and another that implements the full semantics, but has an inefficient running time: $O(n^2)$ per operation, not amortized.

Finally, we analyzed several possible data structures for encoding partial orders for transmission, and representing partial orders as data structures for efficient computation. We concluded that for PO/PR transport protocols the best PO representation both for transmission and internal storage is an adjacency list representation of the transitively reduced precedence graph corresponding to the partial order.

Section 6.8 contains many suggestions for future work in the area of algorithm development for PO/PR transport service; the reader is referred to that section for details. In addition, we provide here some additional comments on some of the other future work suggested in this chapter.

Future work: linear extension selection

This dissertation uses only static selection of the initial sending order for the ReMDoR server. The static algorithm used is a greedy algorithm that incorporates a simple heuristic for providing priority for a single audio stream, based on a static prediction of the available bandwidth. Future work may investigate both more sophisitcated static approaches, as well as dynamic approaches, incorporating any or all of the following:

- results from the real-time systems area in rate-monotonic scheduling
- provision for more than one audio stream
- timing of pause and continue objects
- previous work on dynamic linear extension selection for PO/PR service (Marasli et al., 1996b)

Empirical Observation of Internet Metrics

Some of the arguments in this chapter and elsewhere in the dissertation area based on anecdotal observation of what is “normal” for the Internet. In particular, we have made claims of various strengths about TCP Window Sizes, loss rates, and round-trip delays. Work towards continuous sampling of these quantities is clearly useful, not just for the research in this dissertation, but for the entire protocol design field. The IDMaps project (Jamin et al., 2000) is one example of work currently underway along these lines.