Chapter 2

INNOVATIONS IN TRANSPORT SERVICE ORDER AND RELIABILITY

2.1 Introduction

In this chapter, we present an overview of several innovations related to the handling of order and reliability in transport layer services, and the integration of these features with multimedia synchronization.

This chapter is organized as follows. Section 2.2 reviews the origins of PO/PR service, including an overview of Partial Order Connection (POC), a review of related work, and a summary of the innovations in PO/PR service introduced by the dissertation author in POCv2. This is followed by several sections that focus on each of these innovations in turn:

- Section 2.3 describes the addition of a stream abstraction to partial order service.
- Section 2.4 describes the extension of the POC concept from a single periodic partial order per connection to multiple partial orders per connection.
- Section 2.5 surveys some important ideas and previous work related to multimedia synchronization[2] and provides a motivation for the coarse-grained synchronization support in POCv2, which is described in Section 2.6

[2] We place this section in the middle of the chapter rather than with the other background material because this sequence of presentation provides better continuity to the chapter as a whole.

- Section 2.7 describes the design of a buffer access feature to support integrated layer processing (ILP).
- Section 2.8 discusses the benefits of both tighter integration vs. de-coupling of partial order and partial reliability with respect to each other and proposes new semantics for the PR reliability class.
- Section 2.9 discusses a new feature for unordered/partially-reliable (U/PR) service called *ADN-cancel* that uses *Application Data Names* (ADNs) to give applications more control over reliability of individual messages

The design of POCv2 described in this chapter has been partially realized in the Universal Transport Library (UTL) described in Chapter 3. Section 2.10 provides a summary of the current status of this implementation, distinguishing between the features of POCv2 that have been fully implemented, tested and evaluated in this dissertation, and those which constitute directions for future research. Section 2.11 provides a chapter summary.

2.2 Background: Partially-ordered/partially-reliable (PO/PR) transport service

Partially-reliable services are those which provide something between reliable (no-loss) service and unreliable service. There is much previous work on partially-reliable services based a controlled number of retransmissions: for example, (Gong and Parulkar, 1992, Dempsey 1994, Dempsey et al., 1996, Jacobs and Eleftheriadis, 1997). Our approach to partially-reliable service combines the idea of a controlled number of retransmissions with the notion of *reliability classes*: the idea that an application designates individual objects (messages) as needing different levels of reliability. In this work, we describe three reliability classes: *R* (Reliable), *U* (Unreliable), and *PR* (Partially-Reliable). By specifying the reliability class *R* for a particular object, the application indicates to the transport layer that the object should

be retransmitted an unlimited number of times until it gets to the receiver successfully (or the connection is aborted). By contrast, the application can indicate via the reliability class *U* that an object should be transmitted only once and never retransmitted, or by the reliability class *PR* that an object should be retransmitted, but only up to a point. (The class *PR* is described in more detail in Section 2.8)

In addition to partial reliability, (Amer et al., 1994) introduced the notion of *partially-ordered* service. An application using partially-ordered service defines a partial order, *PO*, over the objects to be communicated, and provides a representation of *PO* to the transport layer. Objects submitted by the sending application may then be delivered to the receiving application in any delivery order that is a *linear extension, LE,* of *PO.* The ordered set *LE* is a linear extension of *PO* if it is a linear ordering of the elements in *PO* such that ($x \prec y$ in $PO \Rightarrow x \prec y$ in *LE*). The basic premise of partially-ordered service is that there are applications that have *some* message sequencing requirements, but can allow other messages to arrive in any of several orders. For these applications, partially ordered delivery may provide less delay, and use fewer memory resources than ordered delivery. The next section provides an example to illustrate the idea.

2.2.1 PO/PR transport service illustrated: the "Screen refresh" example

As a simple example to illustrate the concept of partially ordered transport service, the paper (Amer et al., 1994) describes a "screen refresh" application, as shown in Figure 2.1. In this application, the contents of overlapping windows on a display must be updated from a remote server over a single transport connection. For maximum redraw efficiency, it is desirable that each window should be repainted *before* any window that appears on top of it. If we assume that the contents of each

window are sent as a single Transport Service Data Unit (TSDU), then the placement of overlapping windows determines a partial order for TSDU delivery.[3]

Reading clockwise from the upper left, the four cases shown range from one in which a completely ordered service is required, to one where no delivery order requirements exist between any two TSDUs. The more interesting cases lie in between. For example, the arrangement of windows labeled "B" (upper right) results in a partial order where window (1) must be delivered first, followed by either (2) then (3), or else (3) then (2), with window (4) coming last. Thus, there are two possible delivery orders. The key idea is that if window (2) is lost, with partial order A, we cannot deliver window (3) until (2) is retransmitted. Thus, by providing a partially ordered service instead of a totally ordered service, the transport layer can provide lower delay for window (3). This example illustrates the intuition behind the notion that partial order can be used to provide lower delay, a notion confirmed with empirical data in Chapter 5.

[3] Given the various assumptions (single packet per window, all windows on the same server, a single transport connection) let us stipulate that it is at best debatable whether this problem would arise in practice, say, in a system such as X Windows. However, the purpose here is not to propose screen refresh as a practical application for PO transport service. Rather, the purpose is to illustrate the partial order concept, and for that purpose, this example serves us well. The ReMDoR application described in Chapter 4 provides a more concrete example of an application motivating PO transport service.

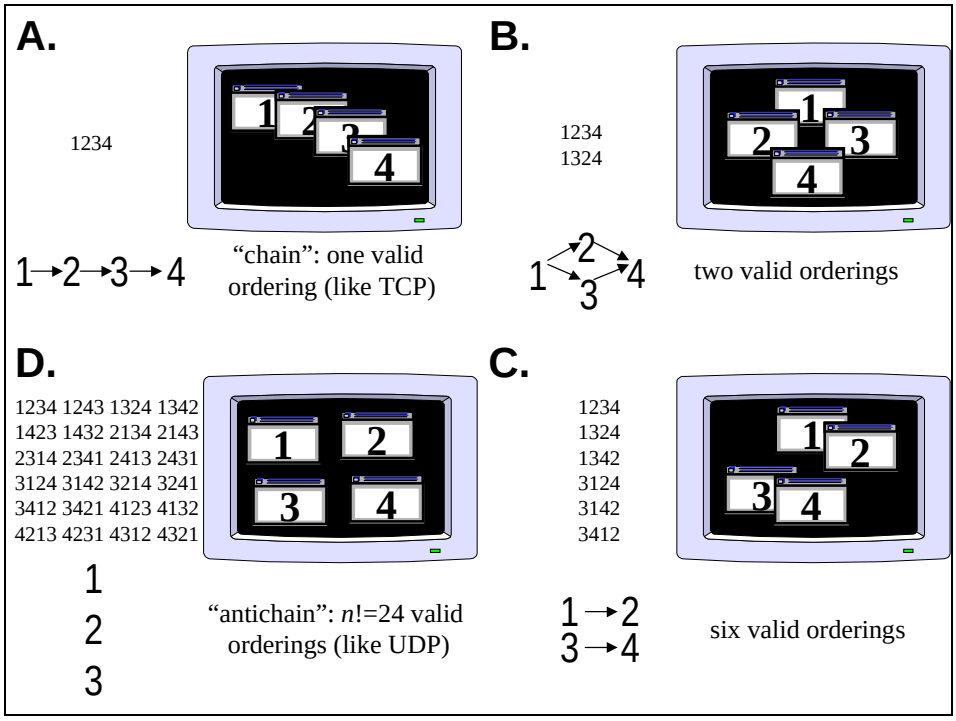

Figure 2.1 Screen refresh example from (Amer et al., 1994)

2.2.2 Notation and terminology related to partial orders

Figure 2.1 provides an opportunity to introduce some notation and terminology that will be used throughout this dissertation.[4] First, the two partial orders representing the extreme cases have special names: the completely ordered case (A), that is the linear order, is a *chain*, while the term *antichain* is used to describe the completely unordered case (D).

For each of the partial orders in Figure 2.1, we show a representation of the partial order as a directed graph. Throughout this dissertation, we make use of the

[4] (Davey and Priestley, 1990) provides a good reference for concepts and terminology related to partial orders.

fact that a directed acyclic graph (DAG) corresponds to a partial order. We thus present diagrams of partial orders where $x \prec y$ in *PO* implies that *x* is drawn to the left of *y*. Since the partial order represents a temporal requirement for delivery, that is, *x* must be delivered *earlier* than *y,* diagrams drawn in this way match our intuition that time flows from left to right across the page. We will typically draw partial orders using the transitively reduced form of the corresponding DAG.

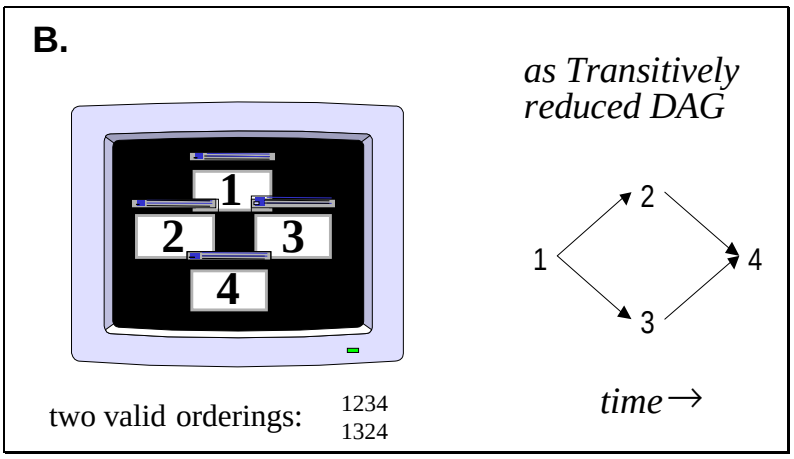

Figure 2.2 Transitively reduced DAG representation of PO

If $\{x \prec y)$ and $(\neg\exists\, v)(x \prec v \prec y\}$, then we say that *x covers y*, or equivalently, that *y is covered by x.* In our diagrams of partial orders, we indicate the cover

ing relationship with a directed edge (drawn as a right pointing arrow as in Figure 2.2) from *x* to *y*. We will also use the term *predecessor* to describe the covering relationship that is implied by the partial order: the *predecessors* of object *k* are all objects $\{ i : i \prec k, \neg\exists\, j, i \prec j \prec k\}$. For emphasis, we sometimes use the term *immediate*

predecessors as a synonym for predecessor. The addition of the adjective immediate does not change the meaning. It only serves to emphasize the fact that the term *predecessor* is reserved for objects that immediately precede a given object in the partial order rather than precede that object through a transitive relationship. The *(immediate) successors* of object k are defined similarly:$\{ i : k \prec i, \neg\exists\, j, k \prec j \prec i\}$.

2.2.3 Previous work on designs for a PO/PR transport service

In Section 1.1 we cited the introduction of POC by (Amer et al 1994), and the simulation and analysis work of (Marasli et al., 1996, 1997a, 1997b)). In addition, Internet RFC1693 describes an attempt to incorporate partial order and partial reliability into TCP (RFC1693). Modifying TCP to provide PO/PR service presents several difficulties. First, partial order and partial reliability are defined in terms of objects (messages), while TCP is fundamentally a byte-stream protocol. Imposing a message framework on TCP alters one of its foundation principles and proves to be cumbersome. Second, the flow control algorithms in TCP are designed based on the assumption of a linear order. Our work with models of POC leads us to believe that using these algorithms with a partially ordered service severely restricts the potential performance improvement that partial order and partial reliability may offer. Third, the need for backward compatibility with TCP makes the design unnecessarily complicated as compared to a PO/PR transport protocol built from scratch. Therefore, for this investigation, we have pursued the development of a new version of POC (POC version 2, or POCv2) designed to make the benefits of partial order and partial reliability available to application developers.

2.2.4 Introduction to Partial Order Connection version 2 (POCv2)

The first version of POC, conceived as an abstract protocol to illustrate the concept of PO/PR service, lacked certain features needed for practical implementation, including facilities for connection establishment and teardown, and a protocol for negotiation of the partial order between sender and receiver—facilities that were introduced in this dissertation work with POCv2. The overall goal in developing POCv2 is to evaluate whether the theoretical advantages of PO/PR services claimed in previous work can be achieved in practice.

2.2.5 Relationship of POCv2 to UTL

The original goal of the software development component of this dissertation was to produce an implementation of POCv2 to serve as the basis of performance study. The end result has been the development of the UTL framework for the development of innovative transport protocols. Thus, although the development of UTL was originally only a means to an end (namely the incremental development of POCv2), in practice, the implementation of POCv2 has been subsumed within the development of UTL. The relationship between POCv2 and UTL is as follows:

- UTL is designed to provide a variety of services, the most general of which is POCv2
- The UTL API and the internal architecture of UTL is designed to support all the features of POCv2 as described in this dissertation

The current implementation supports the subset of those features that are investigated in the performance experiments presented in this dissertation: most especially, partially-ordered/reliable (PO/R) service. The author's future research will include performance experiments to investigate other classes of service, and other

service features: in particular, partially-ordered/partially-reliable (PO/PR) service, and the data preview feature (Section 2.7)

2.2.6 Overview of POCv2 transport service, and comparison with POC

In this section, we summarize the key service features of POCv2. We also outline the key differences between POCv2, and the previous design of POC from (Amer et al., 1994) and (Marasli, 1997b).

- POCv2 is connection oriented and message oriented.
- POCv2 is partially ordered. With respect to partial order, there are several crucial differences between POCv2 and POC.
  - In POC, the partial order had to be defined at the level of individual PDUs. In POCv2, a higher level abstraction called a *stream object* is provided to simplify the specification and processing of the partial order. (Section 2.3)
  - In POC, the partial order was assumed to negotiated in advance by some means external to the transport service. In POCv2, the transport service provides the means to negotiate the partial order. (Section 2.4)
  - In POC, a single partial order governed the entire connection. In POCv2, multiple partial orders can be used, one at a time, over the life of the connection. New partial orders can be negotiated in advance while the connection is in progress. (See Section 2.4)
- POCv2 provides coarse-grained synchronization via the explicit release feature. (See Section 2.6). Synchronization was not considered in POC.
- POCv2 provides a data preview feature to allow coarse-grained integrated layer processing. (See Section 2.7) POC did not provide any such feature.
- POCv2 is partially-reliable. As in POC, a reliability vector provides individual reliability classes for each message. However,

while the semantics of the R and U reliability classes have been retained, the semantics of the PR class have been changed (See Section 2.8).

2.2.7 Service primitives of POCv2: `Read()`, `Write()`, etc.

Rather than provide an exhaustive (but perhaps tedious) description of the complete POCv2 API, in this section we briefly describe the key service primitives of POCv2. Our purpose is to explain just the essential POCv2 concepts and terms that are needed to understand the general descriptions of the POCv2 innovations in the remainder of this chapter, and the POCv2 related material in the chapters that follow.

A POCv2 connection is established with the `Listen()`, `Connect()` and `Accept()` service primitives, which operate exactly as they do in the Sockets API. Once a POCv2 connection is established, a service profile of ordered/reliable service is in effect by default; either application entity may request a change in this service profile through operations described in Section 2.4. The basic functions of input and output in POCv2 are provided by the `Read()` and `Write()` service primitives. A sender uses the `Write()` service primitive to send data. Each `Write()` operation constitutes a single TSDU, which will result in exactly one TPDU being sent across the network. When requested via a `Read()` operation, TSDUs will be delivered as atomic units to the receiver; that is, unlike TCP, POCv2 preserves message boundaries.

The sending and delivery of messages is controlled by the service profile in effect at any given time. The exact interaction between the service profile and the `Read()` and `Write()` operations involves details of the stream abstraction (Section 2.3), the service profile negotiation (Section 2.4) and the explicit release mechanism

(Section 2.6). Therefore, a detailed discussion of this interaction is deferred to those sections.

Once either side of the connection is finished with sending data, the `Close()` operation is used to indicate that that side has no more data to send; this approach is the half-close concept borrowed from TCP. When both sides have done their half-close, the connection is ended.

In addition to the service primitives discussed above, several other service primitives will be introduced later in the chapter. For reference, Table 2.1 provides a list of these with a brief description of each, and the section number where it is introduced.

Table 2.1 A partial listing of POCv2 service primitives

POCv2 service Primitives (also called "operations" or "functions")	**Description**	**introduced in Section**
`Listen()`	passive open: listen for a connection	2.2.7
`Connect()`	active open: establish a connection	2.2.7
`Accept()`	completion of passive open: accept a specific incoming connection request	2.2.7
`Read()`	receive data from the peer application	2.2.7
`Write()`	send data to the peer application	2.2.7
`Close()`	close the connection	2.2.7
`SetSendServPro()`	set the sending service profile	2.4.5
`ReleaseSuccessors()`	release the successors of an object for delivery (used for coarse-grained synchronization of multimedia objects)	2.6.1
`Preview()`	preview out-of-order buffered data (provides for coarse-grained integrated layer processing)	2.7

2.2.8 POCv2 sequence numbers: epoch, period, objNum, cellNum

As background to the detailed sections on POCv2 innovations, we provide a brief overview of key terms and concepts.

Every TSDU sent via the `Write()` operation carries four sequence numbers: an *epoch number*, a *period number*, an *object number* (henceforth: *objNum*) and a cell number (henceforth: *cellNum*).

For example, consider an excerpt from the middle of a multimedia document (illustrated in Figure 2.3). In this document, a single image of the Eiffel Tower is presented in parallel with an audio clip, followed by a text box "Paris at Night." This text is followed by another image of the Eiffel Tower, presented in parallel with another audio clip. Skipping over the epoch number for the moment, we see in Figure 2.4 that in POCv2, this portion of the document can be represented as two periods (shown as periods 7 and 8 in the figure.) Period 7 is a group of two objects: the audio clip (object number 0) and the image of the Eiffel Tower (numbered 1). Period 3 has three objects, the text box, the audio clip, and the nighttime image of the Eiffel Tower (numbered 0, 1 and 2 respectively.)

If we examine the figure at a lower level, we see that the audio and image objects are made up of multiple *cells*. In POCv2, a *cell* is an individual TSDU that is part of a larger object, while an *object* is a single element in the partial order. If the audio clip in period 7 is 5 seconds long, it may be sent in 125 packets, each carrying 1/25 of a second of data (320 bytes for 8Khz μ-law encoding.) Similarly, the image of the Eiffel Tower may be segmented into multiple packets, say 25 packets of 1024 bytes each. Each of these packets is considered a single cell, and is assigned a *cellNum* between 0 and *c*-1 (where *c* is the number of cells in the object). Because each of the

audio cells is part of the same object, each carries the same objNum (zero, in the case of the audio object in period 7.) Similarly, the packets representing the image in period 7 are each assigned an objNum of 1, and a cellnum between 0 and 24. This assigning of cell numbers and object numbers is part of the *stream abstraction* of POCv2 and is described in more detail in Section 2.3.

The period number pertains to the repetition of the service profile. The partial orders used in POC and POCv2 are finite. To allow an arbitrarily large number of objects to be sent over a single connection, both POC and POCv2 allow a partial order to be repeated over multiple *periods.* The rule for multiple periods is that all objects from period i must be delivered before any object from period $i + 1$. Thus, the period number indicates how many times the service profile has been restarted from object 0. The notion of periodic partial orders is described in more detail in Section 2.4.4

Finally, the *epoch* number is introduced in POCv2 to allow for multiple service profiles during a single connection. In POCv2, an epoch is a sequence of zero or more consecutive periods that use the same service profile. When service profiles are negotiated between sender and receiver, they are identified with epoch numbers. Thus, the epoch number on a TSDU or TPDU indicates which service profile should be used to interpret the objNum. Section 2.4.5 provides more detail about the epoch number.

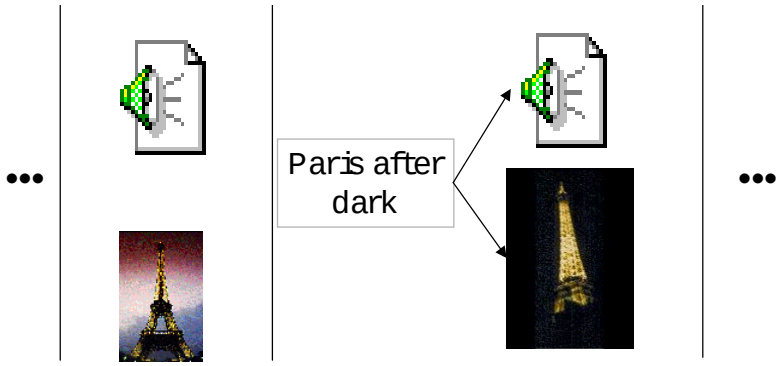

Figure 2.3 Excerpt illustrating POCv2 sequence numbers

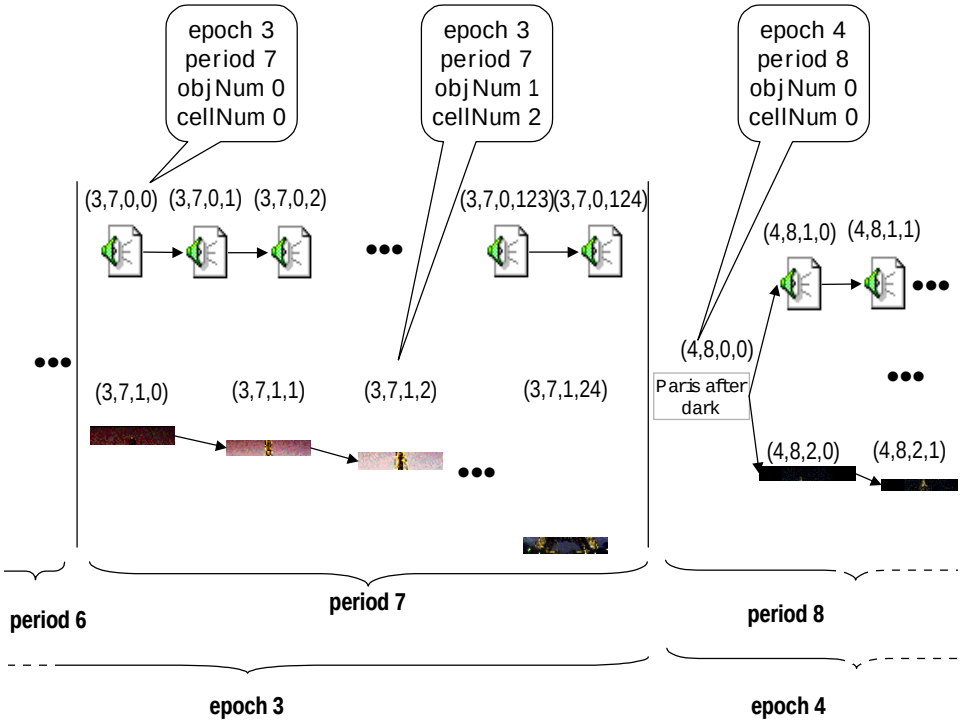

Figure 2.4 Example of epoch, period, objNum, cellNums

2.3 The POCv2 stream object abstraction

In POC, each element in the partial order is assumed to represent an individual message, and corresponds to a single packet on the network. If an Application Data Unit (ADU) is too large to fit into a single packet, it is assumed that the ADU can be fragmented into multiple objects, and these multiple objects can be represented separately in the partial order. However, this is impractical for a simple reason: the resulting partial orders would become too large to be effectively managed by the application, or efficiently processed by the transport protocol. In this section, we present two examples to illustrate this point, and then explain how the *stream* abstraction of POCv2 helps to address this problem.

2.3.1 Motivation for stream objects

First, we consider a small example to illustrate the idea. Figure 2.5 shows a storyboard for an excerpt from the `paris.pmsl` (see appendix.) while Figures 2.6 and 2.7 show two representations of the partial order for this document. Here is an explanation of the excerpt, and the purpose of each of the partial order constraints:

- First, a map of Paris is placed on the screen.
- Then an audio clip (labeled "welcome.au" in Figures 2.6 and 2.7) is played with a narrator saying "Welcome to Paris."
- This is followed by an audio clip ("intro.au") that plays in parallel with three images that are placed overlaying the map. This audio clip says "As you can see on the map, there are many sites in Paris to visit, some of which are more popular than others. On this brief tour, we will see three of the most popular sites." Next, there are three additional audio clips played in sequence, describing each of the three sites.

What makes this document excerpt an ideal illustration of partial order is that

- the narration and the three small images can be presented in parallel, while at the same time,
- there is a constraint that each small image appears completely on the screen before the site portrayed in the image is described by the narrator.

Note this crucial detail in the storyboard. While the Louvre Pyramid is being described, the images of Notre Dame and the Eiffel Tower are not necessarily yet complete. However, *each* of the three images *is* necessarily complete before it is described by the narrator.[5]

Figures 2.6 and 2.7 show two ways that we might represent this partial order. Figure 2.6 represents the partial order at the object level, which is what the stream abstraction of POCv2 provides for,. By contrast, Figure 2.7 represents the partial order at the level of ADUs— that is, at the level of individual packets that would be transmitted across the network as atomic units, as would be necessary using POC. In the next section, we explain why the POCv2 approach is preferable.

2.3.2 Advantages of representing the PO at a higher level than packets

We claim that the partial order in Figure 2.6 is superior for three reasons. The first, and most compelling argument is that when the partial order is represented at the level of TPDUs, the size of the partial order becomes much larger, resulting in unacceptable processing times. For example, in this rather small multimedia document, the size of the PO ($n = |PO|$) grows from n=9 to n=630. For the full `paris.pmsl` document described in the appendix, the PO would grow from n=168 to

[5] In the full `paris.pmsl` document, there are arrows that highlight each location as it is described and indicate its location on the map; these details are omitted in this example.

n=5113. Since the partial order is represented as the adjacency list of a transitively reduced precedence graph, it is often necessary for the application to compute a transitive reduction prior to using POCv2. [6] The fact that the transitive reduction of a DAG is of the same order as matrix multiplication (Aho et al., 1972) presents another important motivation for keeping the a partial order's size as small as possible. the best known algorithms for matrix multiplication are strictly harder[7] than $O(n^2)$, and the algorithm commonly used in practice[8] for transitive reduction is $O(n^3)$. Thus an increase from 168 to 5113 potentially represents an increase in the processing time by a factor of $(5113/168)^3$=28190. The time to compute the transitive reduction and/or the transitive closure for 168 elements is already between 500 and 1000 ms on a lightly loaded Sun Ultra 10. We can therefore expect that the time to compute this value for a partial order of size 5113 would be between 4 and 8 hours (8 to 16 if both computations are done!) Although faster machines would obviously help, and Moore's Law is still with us, regardless of the power of any future hardware, the $O(n^3)$ nature of the algorithm will always provide an incentive to keep the size of the partial order as small as possible.

---

[6] POC uses a transitively closed matrix representation, which presents the same difficulties—more in fact, since the size of the representation of the partial order is $O(n^2)$ for a matrix, vs. $O(n+e)$ for an adjacency list, where e is the number of edges in the DAG.

[7] $O(n^{(2.36)})$

[8] In practice, rather than apply the detailed reduction from transitive reduction to Matrix Multiplication, it is more feasible to reduce transitive reduction to transitive closure, and apply Warshall's algorithm, yielding a running time of $O(n^3)$ (Aho et al., 1972; Cormen et al., 1990.)

Furthermore, consider the representation of the partial order for transmission over the network. POCv2 uses the adjacency list of the transitively reduced precedence graph to send the PO over the network. Granted, a graph such as that portrayed in Figure 2.7 is sparse in terms of edges. Nevertheless, it still has 630 elements; even if it were an antichain, which has the minimum size adjacency list representation, it would require 630 zeros to represent (indicating the zero length of each adjacency list).[9] The graph in Figure 2.6 has only nine elements; because it is transitively reduced, it could have *at most* $(n(n-1)/2) = 36$ edges, resulting in a maximum size adjacency list representation of size $9 + 36 = 47$; which is more than 13 times smaller.[10]

Finally, we argue that representing the partial order at the level of ADUs is a more natural way to represent the partial order for the application. For multimedia documents, it is more natural to think in terms of entire media objects. While this is admittedly speculative, we expect that this would be true as well for other hypothetical applications for partially-ordered transport service. Nevertheless, even if this speculation turns out to be wrong, the arguments related to the algorithms and transmission length would provide sufficient motivation to put some effort into minimizing the PO's size.

---

[9] The total size of the service profile for an antichain of 630 elements would be 1262 if the reliability vector and service profile header were also included.

[10] The total size of the service profile for the maximum size PO over 9 elements (a chain) would be 58 if the reliability vector and service profile were added. Thus the total service profile for a maximum size service profile over 9 elements is actually over 21 times smaller than a minimum size service profile over 630 elements.

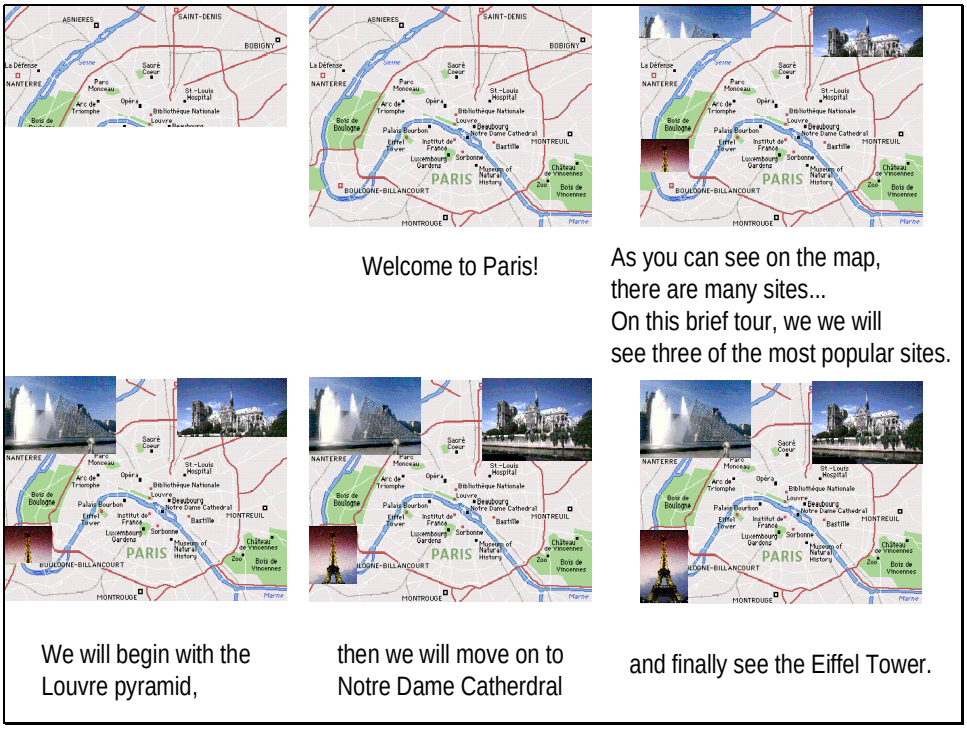

Figure 2.5 Storyboard to motivate POCv2 stream objects

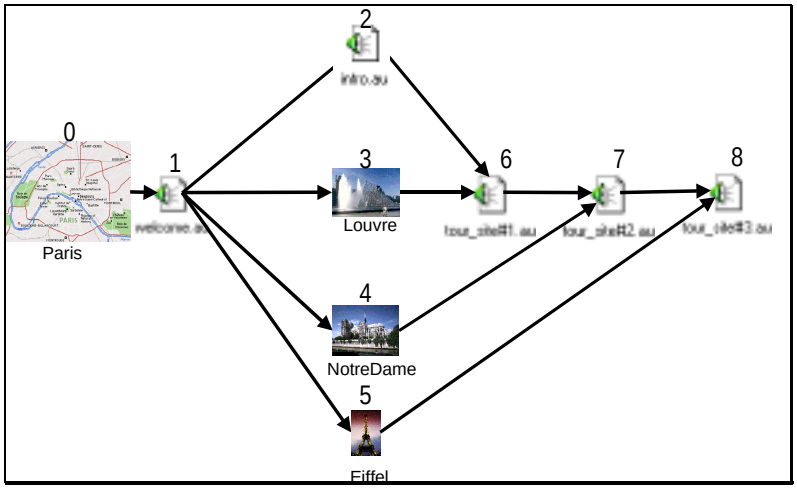

Figure 2.6 Partial order with stream objects

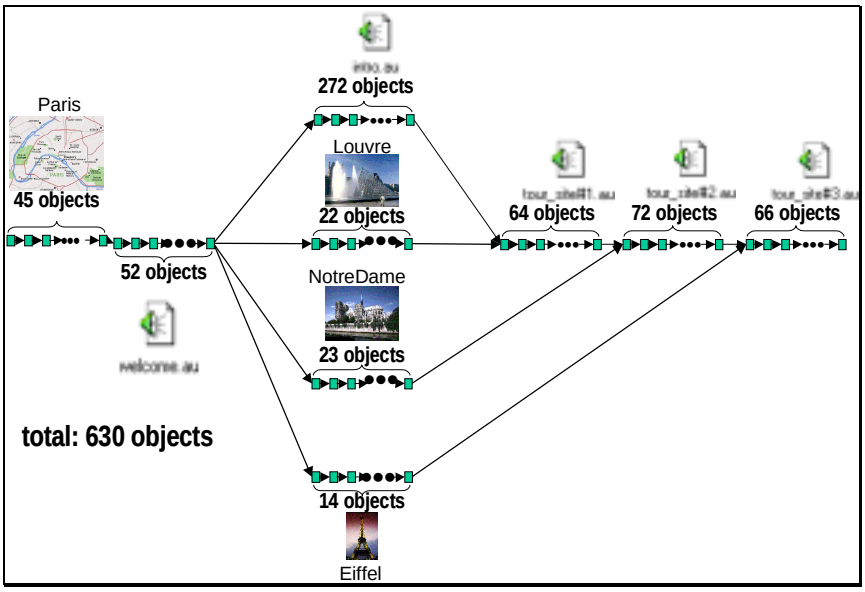

Figure 2.7 Partial order without stream objects

2.3.2 POCv2 stream objects, cells, the streamEnd flag, objNums, and cellNums

The stream abstraction in POCv2 provides the means to use the partial order representation of Figure 2.6, while still getting the semantics of Figure 2.7. For an application using POCv2, every object in the partial order is considered to be a stream of *cells*, each of which is a single packet—more precisely, a single TSDU and single TPDU. The Application Level Framing principle states that the application itself is the best judge of where to segment packets for maximum application efficiency. Therefore, by this principle, it is the application's responsibility to segment an ADU into multiple cells and submit each cell to POCv2 separately. However, for each `Write()`, the application indicates to which object in the partial order the cell belongs, as well as indicating the end of each object. Objects consisting of a single packet (cell) are considered just a special case of stream objects, where the first element in the stream is also the last. POCv2 will then enforce total order

delivery over all cells in the stream of a particular object, but will enforce partial order delivery among cells from separate objects, according the partial order specified at the object level.

Two POCv2 parameters are used to provide the stream object abstraction: these are the *streamEnd* and *currObjNum* parameters. The application can get or set the current value of these parameters at any time. The *currObjNum* parameter is an unsigned integer indicating the object number in the partial order of the object to which the next `Write()` operation will append a cell. The *streamEnd* parameter is a Boolean value: if true, then the next `Write()` operation ends the object being written.

2.4 Service profile management in POCv2

This section describes the features added to POCv2 for management of the service profile, which is the means by which an application using POCv2 specifies its particular order and reliability requirements. Section 2.4.1 describes the limitations of POC in this regard, and motivates the additional service profile management capabilities of POCv2. Sections 2.4.2 through 2.4.5 provide details concerning these additional capabilities.

2.4.1 The need for service profile negotiation and multiple partial orders

In POC, there was only one partial order governing a connection. To change the partial order, a new connection had to be established. In addition, with the exception of RFC1693, in the POC scenarios described or modeled in previous literature, no service primitives or protocol for negotiating the partial order were defined. The partial order was assumed to be known by both sender and receiver in advance.

We claim that this limits the usefulness of POC in practical terms. First, there must be some way for the application to communicate the partial order to both the sending and receiving transport entities. Second, consider the case of using POC for multimedia document retrieval: experience with HTTP 1.0 on the World Wide Web has established that opening a new connection for each document is a bad practice: the case for so-called *persistent connections* to a document server is described in detail in (Mogul, 1995), and has been incorporated into the design of HTTP 1.1 (RFC2068; Fielding et al., 1998).

Thus, two things are needed beyond what is provided in POC: (1) a way for an application to specify a partial order, and (2) a provision for multiple partial orders over the lifetime of the connection. In the sections which follow, we describe how each of these is provided in POCv2.

2.4.2 POCv2 service profiles: notation, formal definition, and representation

In this section, we introduce a notation and formal definition for a POCv2 service profile based on the notation for service profiles introduced in (RFC1693). We then discuss the problem of representing a service profile in a bit string for communication over the network.

As shown in Figure 2.8, a POCv2 service profile is formally defined as a tuple $SP=\langle n, PO, RV\rangle$, where:

- n is a non-negative integer, representing the cardinality of a set E of objects (elements) to be communicated from sender to receiver, i.e., $n = |E|$.
- PO is a partial order over the set {0, 1, 2 .. n-1}. This partially ordered set is mapped one-to-one and onto the set E, and constitutes the *object numbers* of the elements of set E.

- *RV* is a length *n* reliability vector over the set of reliability classes *RC* = {*U*, *R*, *PR*}, i.e., $RV \in \{RC\}^{n}$;

  $RV = \langle rc_0, rc_1, rc_2, \ldots rc_{n-1} \rangle$, where rc_i represents the reliability class of object *i*.

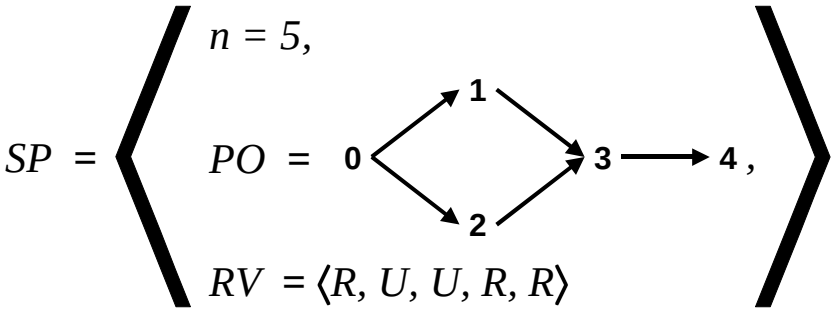

Figure 2.8 Example definition of a POCv2 service profile

2.4.3 Representation of the service profile

For purposes of transmission over the network, or storage on a disk file, a POCv2 service profile is represented as an array of unsigned integers that encode the reliability vector, and the transitively reduced adjacency list of the partial order. For example, the partial order and reliability vector shown in Figure 2.8 would have the following array representation[11]:

[11] The line breaks in the example above are inserted only to assist with human interpretation; the integers would be stored consecutively in an array.

```
17 5
0 2 1 2
1 1 3
1 1 3
0 1 4
0 0
```

The interpretation of this array is shown in the following diagram:

ℓ n
rc_0 $numSuccessors_0$ {*successor, successor ... successor*}
rc_1 $numSuccessors_1$ {*successor, successor ... successor*}
...
rc_{n-1} $numSuccessors_{n-1}$ {*successor, successor ... successor*}

where:

- ℓ is the length of service profile (the total number of integers in the encoding, including ℓ itself.)
- *n* is the number of elements
- rc_i is the reliability class of element *i*, where:

  0: R, reliable

  1: U, unreliable

  2: PR, partially reliable

- $numSuccessors_i$ is count of successors that follow (may be 0)
- {*successor, successor, ... ,successor*} is a list of successors of element *i* in the partial order. The total length of this list should equal $numSuccessors_i$.

2.4.4 Periodic partial orders in POC

POCv2 builds on the notion of periodic partial orders that we previously defined in POC; therefore in this section we review this concept. As explained above, in POC, only one partial order is permitted per connection. One might therefore be led to believe that only $n=|PO|$ objects could be sent over a given connection. To allow for a larger number of objects, POC allows the partial order to be repeated in periods. The top half of Figure 2.9 illustrates this idea. The figure shows a single partial order with 6 objects numbered 0 through 5, that repeats through multiple periods (the first five periods, numbered 0 through 4 are shown.) Thus each TSDU (and hence each TPDU) actually carries two sequence numbers: the *object number* (henceforth referred to as the *objNum*) and the *period* number.

The semantics of the resulting *periodic partial order* are as follows:

- for all $i \leq 0$, all objects from period p_i must be delivered before any object from period p_{i+1}, and
- within a period, all object delivery respects the partial order; that is, all objects $e_i \prec e_j$ must be delivered before object e_j

While we focus here on the partial order, in fact the entire service profile (including the reliability vector) repeats in the same fashion.

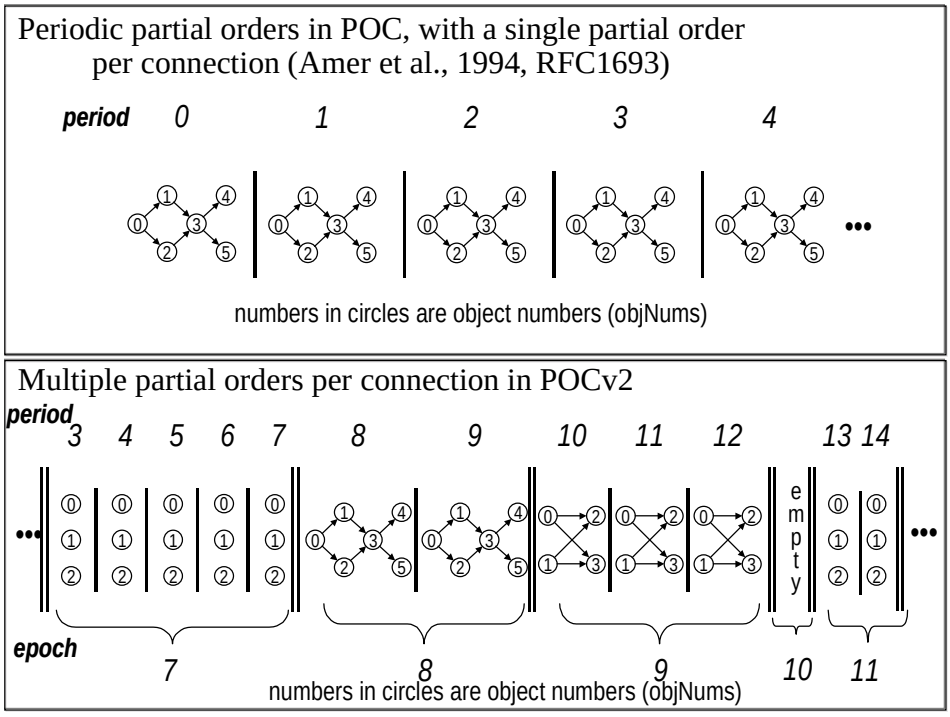

Figure 2.9 Periodic partial orders in POC, POCv2

2.4.5 Multiple periods and epochs in POCv2

In POCv2, we extend the concept of periodic partial orders. In POC, each TSDU/TPDU carries only two sequence numbers (*period*, *objNum*). In POCv2, we add the *epoch* numbers and the *cell number* (*cellNum*). The cellNum was described previously in Sections 2.2.8 and 2.3.2. In this section, we describe the use of the epoch number.

As shown in the bottom half of Figure 2.9, in POCv2, an *epoch* is a series of zero or more consecutive periods that all have the same service profile. For example, in the Figure 2.9, epochs 7, 8, 9, 10 and 11, contain 5, 2, 3, 0, and 2 periods, respectively. The first epoch in a connection is always numbered epoch 0, and by

default is defined to have the default POCv2 service profile, which consists of a single reliable object per period. The array representation for this default POCv2 service profile is shown below:

```
4 1
0 0
```

When the connection is first established, this default service profile is in effect for each side of the connection. For each direction of the full-duplex POCv2 connection, the sending side manages the service profile. To change the service profile for a given direction, the sending application for that direction issues the `SetSendServPro()` operation. This operation adds a new epoch to the list of epochs that may be used for the data flow from service user issuing the `SetSendServPro()`operation, to the peer service user. The first `SetSendServPro()` operation defines the service profile for epoch 1, the next call defines epoch 2, and so forth.

Each time a new epoch is created in this manner, the service profile is communicated via POCv2 control TPDUs from the POCv2 sender to the POCv2 receiver. These TPDUs are sent reliably, and retransmitted if necessary. Thus, the POCv2 transport entities on both sides of the connection eventually learn the service profile for every epoch defined by the sending application for a given direction of data flow.

Note that while object numbers start from 0 with each new period, this is not the case for period numbers. That is, period numbers do not start over from 0 with each new epoch. A period is *bound* to a given epoch when the first data TPDU arrives containing both a period number and an epoch number.

The sender moves from one epoch to another anytime the sequence of `Write()` operations leaves the connection at a *period boundary*. The connection is at

a period boundary at connection establishment time, and again at anytime when every object in the current partial order has been finished. (That is, the last cell in each object has been written via a Write operation with the *streamEnd* flag set to true, as described previously in Section 2.3.2).

Any time the connection is at a period boundary, the sending application can perform the `IncrEpoch()` operation to increment the epoch. This operation leaves the connection still at a period boundary, but with a new partial order in effect.

For example, if an application wants to override the default service profile of ordered/reliable service in effect by default for epoch 0, the application issues the following sequence of operations: first, a `SetSendServPro()` operation to set the service profile for epoch 1, then an `IncrEpoch()` operation to move to epoch 1 before the first cell is written. Similarly, a series of two `IncrEpoch()` operations with no `Write()` operations in between skips over an epoch entirely; for example, this is how period 10 in Figure 2.9 ended up with zero periods.

2.5 Multimedia synchronization: background material

One of the key proposals of this dissertation is that there are benefits to integrating coarse-grained synchronization with PO/PR transport service. Thus, some brief background information concerning multimedia synchronization will be helpful. First, we distinguish between coarse and fine-grained synchronization. Second, we provide a brief synopsis of Pérez-Luque and Little's *Temporal Reference Framework for Multimedia Synchronization* (Pérez-Luque and Little, 1995), which provides a unified theoretical foundation for various methods of specifying temporal scenarios. Finally, we focus on one such method: namely, the Object Composition Petri Net (OCPN) of Little and Ghafoor (Little and Ghafoor, 1990). OCPN is of particular

interest, because it provided the initial motivation for applying the Partial Order transport concept to multimedia applications (see Amer et al., 1994).

2.5.1 Coarse-grained vs. fine-grained synchronization

Discussions of multimedia synchronization often distinguish between two levels of synchronization: *coarse-grained synchronization* and *fine-grained synchronization* (Wynblatt, 1995; Schnepf et al., 1995.)

Coarse-grained synchronization (also called *temporal alignment*) refers to synchronizing the start and end of objects with respect to one another. For example, consider a document describing a travelogue of Paris (`paris.pmsl` in the appendix), containing (among other things) a map of the city, pictures of several attractions, and audio describing these attractions. The requirement that the map be on the screen before the audio begins would be an example of a coarse-grained synchronization requirement. Other examples might be the requirement the pictures of the attractions appear at some point during the audio description, and/or that arrows pointing to specific attractions occur at specific points during the audio description.

Coarse-grained synchronization can be distinguished from *fine-grained synchronization* (also called *stream synchronization*) which refers to keeping parallel streams synchronized with one other (as in lip-sync, for example) (Wynblatt, 1995; Schnepf et al., 1995.) Fine-grained synchronization would come into play in our Paris document if there were a so-called *talking head*—that is, a video of a human announcer describing the Paris attractions. In terms of transport layer design, and the design of transport/application layer interactions, fine-grained synchronization is of

little interest, because there is precious little that the transport layer can do to assist. Fine-grained synchronization mostly pertains to issues of operating system scheduling of threads that control the video and audio hardware on the end system. From the networking perspective, the best we can do is to ensure that the video data and audio data that need to be presented simultaneously should be encapsulated in the same TPDU. Indeed, even when audio and video data are coming from a local disk, synchronized audio and video are generally treated as a single interleaved data stream (Szabo and Wallace, 1991). This approach is used, for example in MPEG-1 (Le Gall, 1991).

On the other hand, where coarse-grained synchronization is concerned, networking issues—particularly, transport layer issues—definitely come into play. Indeed, one of the key proposals of this dissertation is that there are benefits to integrating coarse-grained synchronization with PO/PR transport service. Thus, some brief background information concerning coarse-grained synchronization is warranted.

2.5.2 Temporal scenarios for coarse-grained multimedia synchronization

(Pérez-Luque and Little, 1995) provides a survey and taxonomy of various methods of specifying and enforcing temporal scenarios for coarse-grained synchronization. One crucial distinction is between *determinate* and *indeterminate* scenarios. Determinate scenarios are those in which events occur at specific times, such as “35 seconds after the beginning of the document”, or “06/11/99 at 03:45:17 hours UTC.” These scenarios have the advantage of simplicity, are most useful for situations where there is little to no probability that an object will be unavailable at its scheduled time of presentation. However, they do not allow for variations introduced by human interaction with the document, nor do they allow for variation in network QoS.

More useful for a networked application are schemes for specifying *indeterminate scenarios*. In these schemes, instead of specifying specific times at which events occur, relationships among various events are expressed, allowing for possibly many different realizations of the event timeline, any of which would be acceptable to the document author.

Another distinction is made between the unit of expressing time in the temporal scenario. There are three possibilities:

- *dates*, which are specific determinate points in time, such as 00:35.14 seconds,
- *instants*, which are indeterminate points in time, such as "the instant that the audio starts playing", and
- *intervals*, which are sets of instants lying between an start and end instant, such as "the interval during which the audio is playing"

Finally, (Pérez-Luque and Little, 1994) distinguishes between *quantitative* and *qualitative* specifications. Quantitative information is "temporal information that can be expressed in time units (e.g., t_1 = 6PM, or the length of the interval [a, b] is 3 hours.)". Qualitative information, on the other hand is "temporal information that is not quantifiable", such as total and partial orderings of instants and/or intervals. For example, given any two instants or dates *a* and *b*, one of three relationships holds: either a occurs *before* b, *after* b, or *at-the-same-time-as* b. Of particular interest for this dissertation are the thirteen possible *binary temporal relationships* between intervals.[12] Figure 2.10 illustrates seven of the thirteen relations; the other six are the

[12] Most authors credit (Hamblin, 1971) as the first reference for the thirteen binary relationships between time intervals. As this paper may not be widely available, some more useful references on this topic may be (Allen, 1983) and (Little and Ghafoor, 1990).

inverses of the seven shown (e.g., *before*$^{-1}$, *meets*$^{-1}$, etc.), with the exception of the equals relation, which has no inverse. These intervals are often used as a basis for proving that a given temporal specification scheme has sufficient power to represent all possible relationships among atomic processes, for example, see (Little and Ghafoor, 1990; Hoepner 1991).

2.5.3 Object Composition Petri Nets (OCPN)

With these definitions, Pérez-Luque and Little then categorize several important multimedia systems from the research literature according to this taxonomy. One system of particular interest is the *Object Composition Petri Net* (*OCPN*) (Little and Ghafoor, 1990). OCPN is a scheme for multimedia synchronization based on Timed Petri Nets. In this dissertation, OCPN serves as the basis for characterizing both the power and limitations of the explicit release synchronization feature of POCv2, therefore this section summarizes the key definitions, notations and properties of the OCPN model necessary for that purpose.

Figure 2.11 illustrates the idea of OCPN, using a slide presentation as an example. A basic[13] Petri net $N=(T, P, A)$ consists of a set of *transitions* (indicated by vertical bars), a set of *places* (illustrated by circles), and a set of directed *arcs*. In the OCPN, each place in P represents a media object to be presented, such an audio clip, or a still image. In an OCPN: $C= (T, P, A, D, R, M)$:

- *T*, *P* and *A* are defined as in the basic Petri net.

[13] A more common formulation of the basic Petri Net is C=(P,T,I,O), where P is the set of places, T is the set of transitions, and I and O represent functions mapping transitions to sets of input and output places respectively (Peterson, 1977). We instead follow the lead of (Little and Ghafoor, 1990) and use the N=(T,P,A) formulation found in (Agerwala, 1979).

- *D* maps each place to a duration (e.g., the time it takes to present a certain object)
- *R* maps each place to a resource (e.g., the hardware resource needed to present the media, such as the speaker, or certain portion of the display),
- *M* indicates a marking of tokens (dots) to each place.

OCPNs are composed using subnet replacement based on Petri nets representing Hamblin's thirteen binary relations between intervals (shown in the right half of Figure 2.10). A key consequence of this construction is that each place in an OCPN has exactly one incoming and one outgoing arc—a property we will make use of in Section 2.5.4.

As an example to illustrate the connection between the OCPN and multimedia documents, Figure 2.11 shows a document where a series of slides (still images) is presented in parallel with a series of audio clips. After each slide, a point is reached when the audio and image must be synchronized before moving on to the next slide. In an OCPN, this is modeled by the following sequence of events. Initially the first transition is fired to start the presentation. The transition rules are then summarized in (Little and Ghafoor, 1990) as follows:

> "(1) A transition fires immediately when each of its *input places* [the places from which an incoming arc is directed to the transition] contains an *unlocked* token.
>
> (2) Upon firing, the transition … removes a token from each of its input places, and adds a token to each of its *output places* [that is, each place to which there is an outgoing arc from the transition.]
>
> (3) Upon receiving a token, a place remains in the active state for the interval specified by the duration τ_j [determined by the duration mapping D: $P \rightarrow \Re$]. During this time, the token is *locked*. [and the media object is playing]. When the place becomes inactive

> [i.e., the object is finished playing], or on the expiration of the duration τ_j, the token becomes unlocked."[14]

Thus, the notion of *locking* and *unlocking* of the token based on object durations is key to the OCPN model of synchronization. We will revisit this locking/unlocking idea when we discuss the benefits and limitations of integrating a PO/PR transport service with coarse-grained multimedia synchronization via POCv2, UTL and ReMDoR.

(Little and Ghafoor, 1990) includes the result that an arbitrarily complex process model composed of Hamblin's thirteen temporal relations can be constructed with OCPNs by choosing pairwise, temporal relationships between process entities. The right half of Figure 2.10 captures the main idea of this proof, as it illustrates the OCPN constructions that are used to model each of Hamblin's intervals. OCPNs are thus shown to be a powerful tool for specifying temporal scenarios for multimedia synchronization. (In Section 2.6.4, we show how this result can be extended to explicit release synchronization in POCv2 under the ideal conditions of a perfect channel. We then describe the impact of network delays on explicit release - synchronization.)

On the other hand, OCPNs have limitations. (Pérez-Luque and Little, 1995) explains that OCPN cannot express indeterminate temporal scenarios, in spite of the fact that OCPNs are based on qualitative, rather than quantitative, relationships between intervals. This limitation is a consequence of the fact that the locking and unlocking of tokens is based on fixed (i.e., deterministic) object durations. (Little and

---

[14] For completeness, we note that while place i is active, the place makes uses of the resource specified by the resource mapping R: $P \rightarrow \{resource_1, resource_2, \ldots, resource_k\}$. However this is not an issue we will address further in this dissertation.

Ghafoor, 1990) acknowledges that OCPNs cannot capture user interaction, or VCR-like-features such as rewind.

2.5.4 Extended OCPN (XOCPN)

Later work by (Woo et al., 1994) on the Extended OCPN (XOCPN) describes a technique for sending OCPN data over an ATM network. The authors describe how the durations in an XOCPN can be used to back-calculate the necessary sending times to ensure timely delivery of document objects. The goal is to have objects that are to be presented at the same instant arrive at the destination host at the same instant. However, Woo et al.'s work addresses a fundamentally different problem from that addressed in the current dissertation: Woo et al. assumes a high-bandwidth, low loss ATM network with QoS guarantees at the network level, and asks how one can best make use of this network. By contrast, in this dissertation, we are interested in providing the best service possible when the network provides no QoS guarantees whatsoever, or where we have to make do even when the network layer "breaks its promises". Thus we now turn to a discussion of the first innovation introduced in this chapter: the integration of multimedia synchronization with PO/PR transport service.

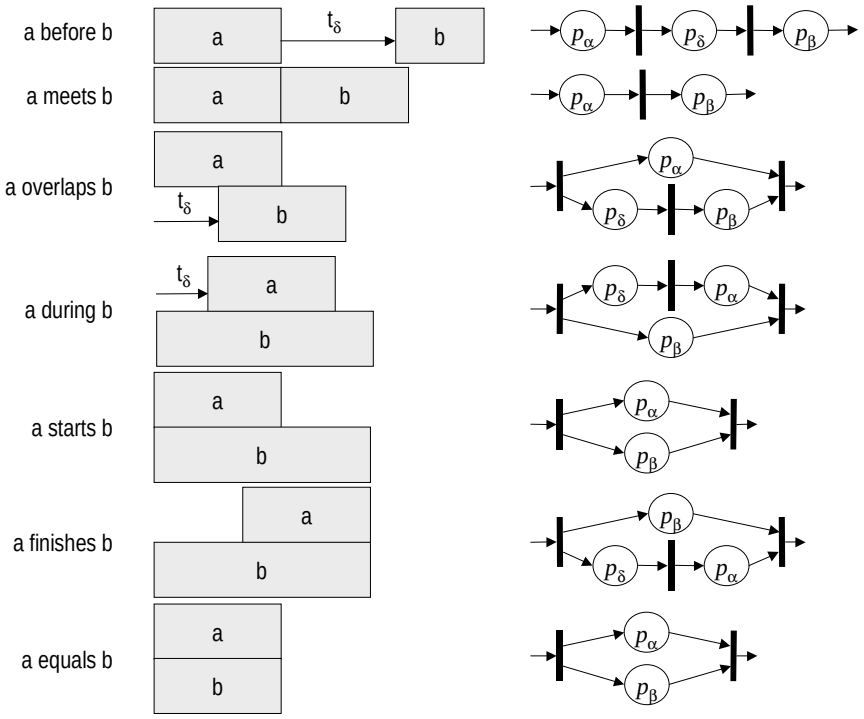

Figure 2.10 Hamblin's temporal relations (left); corresponding Petri nets from OCPN (right) (Little and Ghafoor, 1990)

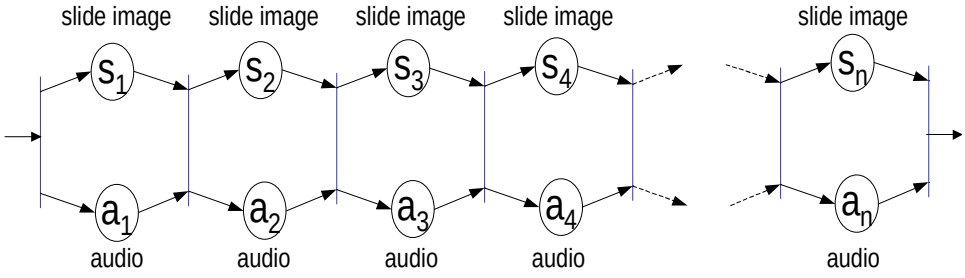

Figure 2.11 Example OCPN for slide presentation adapted from (Little and Ghafoor, 1990)

2.6 Transport layer support for multimedia synchronization in POCv2

POCv2 provides support for coarse-grained synchronization via a feature called *explicit release*. In this section, we outline the operations of this feature, and the advantages and disadvantages of implementing coarse-grained object synchronization in this manner.

```
while (not end of document)
{
   read data from the transport layer
   display that data
}
```

Figure 2.12 Pseudocode for a "read and display" loop

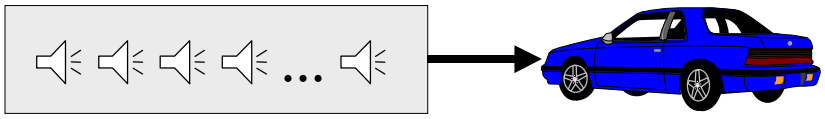

Figure 2.13 Example: two element partial order

2.6.1 Motivation

To motivate the need for coarse-grained synchronization, consider what would happen if we ignored synchronization concerns altogether. For example, consider a naive multimedia application that simply uses a "read and display" loop, with pseudocode as shown in Figure 2.12. Now suppose that our document contains an image that is to appear immediately following the completion of an audio clip. Thus, the image immediately follows the audio clip in the partial order as represented

by the graph in Figure 2.13. Further, assume that the audio clip consists of several packets (a likely assumption, since every second of audio typically requires 8000 bytes at normal encoding rates.) As soon as each packet arrives, the application will end up writing each packet to the audio device for playback.

It turns out that most audio devices keep a playout buffer to store packets in the event that they arrive faster than the playout rate. If there is a backlog in the audio output buffer, then when the last byte of audio is written to the device, it may yet be several seconds before the audio data is finished playing.

Without a synchronization mechanism, the image becomes deliverable immediately upon delivery of the audio clip. The receiving application may then read the image and present it before the playback of the audio is complete, in spite of the fact that the image was intended to *follow* the audio clip in the presentation order. Note that the application cannot simply wait until the audio clip is complete to read the next object from the transport layer, since the next object might be something that is to be presented in parallel with the audio clip.

Thus, unless the client application is aware of the partial order and implements its own functionality to preserve synchronization relationships, such relationships may be violated. The client cannot simply read objects from the transport layer and display them as they become available. Presenting the image as soon as it is deliverable may result in its being placed on the screen before the audio is finished, thus violating the temporal scenario specified by the document author.

Therefore, we can see that some form of synchronization mechanism is needed. The question then becomes: “where should this synchronization be implemented, and how?” As a transport layer designer, the first clear answer seems to

be: “Put this functionality into the application. The transport layer’s job is to just get data to the application as soon as possible, and then let the application work out any synchronization concerns.” Indeed, early work on POC was very clear on this point: “It is assumed that synchronization concerns in presenting the object after delivery is [sic.] a service provided on top of the proposed partial order service.” (Amer et al., 1994).

However, the suggestion to disregard synchronization at the transport layer misses an opportunity that becomes clear when one considers the actions that a partial order protocol takes to ensure that object delivery respects the partial order. It turns out that a minor modification of the transport service API can provide helpful support for multimedia synchronization.

2.6.2 Coarse-grained synchronization via explicit release

The *explicit release* feature of POCv2 involves a small change to the semantics of partial order delivery. Under normal partial order delivery rules, an object is considered deliverable when all of its predecessors have been delivered. By contrast, when explicit release is used, the rule is that an object is deliverable when all of its predecessor objects have been delivered, *and completely displayed by the application*.

To make this change in semantics, all that is necessary is to provide the application a way to explicitly signal the transport layer each time the processing of a delivered object is completed, thereby *releasing the successors* of the delivered object. The POCv2 service primitive `ReleaseSuccessors()`is used for this purpose. If the application chooses to enable explicit releases for a given connection, then for each object that the application reads from the transport layer, the application has the

responsibility to perform a `ReleaseSuccessors()` operation when all the cells for that object have been processed.

In the case of the previous example of an audio clip followed by an image (Figure 2.13), the client simply waits until the audio clip is finished to explicitly signal the release of the audio clip's successors in the partial order. Thus, the application can be assured that the transport layer will not deliver these successors until the time that they should be presented. Section 2.6.3 provides a more formal definition of an explicit release mechanism, while Section 2.6.4 compares its power to other temporal scenarios.

Being able to rely on the explicit release feature for coarse-grained object synchronization removes this burden from the application programmer. The code for a multimedia client can be made simpler if the client can rely on the transport layer to provide basic synchronization support. The exchange of the partial order between sender and receiver happens entirely within the transport layer.

The remarkable consequence is that a multimedia document retrieval *client* using POCv2 can present a document in compliance with a given temporal scenario for synchronization *without ever having to process that specification at all.* Instead, the server simply reads the partial order as part of the document specification, and requests that the transport layer use it for object delivery. The client's only responsibility is to signal explicit release when the playback of each object is complete

2.6.3 Formal definition of explicit release synchronization

Our formal definition of explicit release synchronization begins with an outline of our basic assumptions. These assumptions are expressed in terms that generalize the problem beyond multimedia synchronization, to any computation where

synchronization is necessary, while parenthetically noting the connection to multimedia documents.

Basic assumptions:

- A finite set *E* consists of objects that are to be transmitted from sender to receiver. (*E* may represent a set of multimedia objects; e.g., each element of *E* may be a audio clip, a graphic image, an explicit pause, or an interaction point).

- Associated with each object is some finite amount of processing time, possibly zero[15]. The processing time may be known *a priori*, or may be indeterminate (e.g., in the case of an interaction point in a multimedia document.)

- For simplicity, we assume that all other processing time at the application is negligible, and that each object has a dedicated processor.[16] As a result, the application is always ready to receive objects that the transport layer is ready to deliver, and the processing of each object starts immediately upon delivery of the object.

- The receiving application requires that the processing of each object takes place in specific synchronization relationships with the processing of other objects. The required relationships may be expressed as determinate binary temporal intervals between objects in the manner of (Hamblin, 1971), or as indeterminate binary temporal intervals, as in (Pérez-Luque and Little, 1996) (See Section 2.4.2, Figure 2.10)

Notation and Definitions: For object e_i

- $del(i)$ is the instant of delivery of e_i by the POCv2 entity

[15] Zero indicates negligible processing time. In reality, all objects have some non-zero processing time.

[16] This assumption is not altogether unrealistic if one considers that the most significant processing of audio and video (the actual real-time playout or display) is often performed by dedicated hardware devices.

- *rel*(*i*) is the instant of the `ReleaseSuccessors()` operation on object e_i

- The *active interval* of e_i is the time interval [*del*(*i*),*rel*(*i*)].

Formal Definition of Explicit Release Synchronization:

- A partial order *PO* is defined over the set of objects *E*

- Associated with each object e_i in *E* is a counter c_i, which counts the number of *unreleased predecessors* of object e_i . The value is initialized thus: $c_i \leftarrow |\{e_j: e_j \text{ covers } e_i\}|$. (That is, c_i is initialized to the number of immediate predecessors of e_i, or equivalently, the in-degree of e_i in the transitively reduced DAG corresponding to the PO.)

- For each e_i in *E*, the delivery instant *del*(e_i) is the earliest time at which both the following are true: (1) c_i equals zero, and (2) e_i is available at the POCv2 receiver (i.e., e_i has been successfully transmitted across the network).

- The application performs the `ReleaseSuccessors()` operation at the conclusion of the processing time for each object e_i.

- The `ReleaseSuccessors()` operation is defined as follows:

  foreach (e_j: e_i covers e_j) **do**

    decrement c_j;

In the next section, we use this formal definition to compare explicit release synchronization with Little and Ghafoor's OCPN.

2.6.4 Comparison of explicit release synchronization with OCPN

First we show that under the assumption of a perfect channel, explicit release has as much expressive power as OCPN—that is, for any temporal scenario expressed in OCPN, there exists an equivalent scenario using explicit release. We show this result by following the outline of the proofs from (Little and Ghafoor, 1990) that illustrate the expressive power of the OCPN.

Theorem 2.1 Under the assumption of a perfect channel (no loss, sufficiently small delay that no object arrives later than the time it is needed), an arbitrarily complex process model composed of temporal relations can be constructed with an explicit release temporal scenario.

Our Theorem 2.1 is patterned after Theorem 2 of (Little and Ghafoor, 1990), which shows the same result for OCPN instead of explicit release. Our proof piggybacks off theirs: we first prove a lemma that under the perfect channel assumption, we can go from an OCPN to an equivalent explicit release temporal scenario. An abbreviated version of this lemma was given in (Conrad et al., 1996), however this is the first publication of the full proof. We can then use the Theorem 2 result from (Little and Ghafoor, 1990) and the proof of Theorem 2.1 is immediate. Later, we explore what happens when the perfect channel assumption is relaxed.

Lemma 2.2: Given a temporal scenario for a set of multimedia elements E expressed by an OCPN $C=(T,P,A,D,R,M)$, we can construct a temporal scenario using explicit release that will provide equivalent semantics, assuming a perfect channel (no loss, and sufficiently small delay that no object arrives at the receiver later than the time it is needed in the temporal scenario.)

Proof: We prove Lemma 2.2 by first providing an algorithm to construct the POCv2 service profile that is equivalent to the OCPN, then explaining why the two are equivalent.

Notation: Since P, the set of places in the OCPN, maps one-to-one and onto the set E of multimedia documents elements, we will use the notation: $e_i \in$ E to indicate the element corresponding to place $p_i \in P$.

1) Define a service profile $SP=\langle n, PO, L\rangle$ by defining

- $n = |E|$, or equivalently, $n = |P|$
- PO is a partial order over the objects in E, (for example, representing the elements of the multimedia document), initially an anti-chain.
- L is arbitrary reliability vector (the reliability class of objects does not matter given the perfect channel assumption)

2) Add additional pairs ($e_j \prec e_i$) to the partial order as follows:

for each place $t_i \in T$ **do**
  for each arc $(p_j, t_i) \in A$ **do**
    for each arc $(t_i, p_k) \in A$ **do**
    {
      add $e_j \prec e_i$ to PO
    }

This construction results in modeling each transition with a set of partial order constraints we informally call a “shoelace” for reasons that should be clear from Figure 2.14.

To demonstrate that the explicit release scenario is equivalent to the OCPN scenario, it now suffices to show that interval during which each place $p_i \in P$ is active in the OCPN corresponds exactly to the active interval of each e_i in the resulting explicit release scenario.

In the OCPN, the active interval of any place p_i can be defined as $[f(t_i), f(t_i+\tau_i)]$, where $f(t_i)$ represents the firing time of the transition t_i that preceeds p_i .(Recall that in an OCPN each place has only a single incoming arc, thus t_i is unique, and that τ_i represents the duration of the process modeled by p_i).

In the corresponding service profile, each transition is modeled by a shoelace of constraints. The active interval of the element e_i is $[del(e_i), del(e_i + \tau_i)]$. Since a perfect channel is assumed, $del(e_i)$ occurs at

the instant when the number of unreleased predecessors c_j becomes zero. But, by virtue of the construction in step 2, c_i represents at all times the number of places with incoming arcs to transition t_i that have not yet provided an unlocked token enabling transition t_j to fire. Thus c_i will become zero precisely at the instant that transition t_j fires.

Therefore, since $f(t_j) = \text{del}(e_i)$, we have:

$$\begin{aligned} \text{the active interval of } p_i &= [f(t_j), f(t_j+\tau_j)] \\ &= [\text{del}(e_i), \text{del}(e_i+\tau_j)] \\ &= \text{the active interval of } e_i \end{aligned}$$

and the equivalence of the temporal scenarios is demonstrated. ❑

So given a perfect network we can model all of the determinate temporal scenarios modeled by the OCPN. The situation is different if network delays due to propagation, queuing, insufficient bandwidth, or the necessity of retransmission cause one object in a binary relation to be arbitrarily delayed. Using the framework of (Perez-Luque and Little, 1996), we can characterize the effect of network delays as a replacement of a basic binary relation with an indefinite binary relation.[17] Figure 2.15 shows that only the *before* and *before*$^{-1}$ relations can be precisely preserved. The *meets* and *meets*$^{-1}$ relations are approximated by the indeterminate relations (*meets-or-before*) and (*meets*$^{-1}$*-or-before*$^{-1}$). The rest devolve into disjunctions of relations so long as they in essence constitute no temporal constraint at all: these relations exclude only the basic relations that would be prohibited by inconsistent durations (for example, *equals* can never become *starts*.) It should be noted that OCPN fares no better at modeling temporal relations when burdened with unpredictable network delays.

[17] The set of indefinite binary relations is defined in (Perez-Luque and Little 1996) as the set of all possible disjunctions of the determinate binary relations; for example, "*meets-or –before*, "*equals-or-finishes-or-during*". Thus, there are 2^{13} indefinite binary relations on temporal intervals.

Thus, when loss is introduced, explicit release can preserve (or approximate) only the *meets* and *before* relations, and their inverses. While perhaps disappointing, this result should not come as too much of a surprise; after all, the *meets* and *before* relations constitute the essence of partial order delivery semantics.

Future work: two-color Petri net delivery semantics.

To extend the range of the explicit release approach, we propose as future work, the investigation of a modified *two-color Petri net* delivery semantics that would supplement partial order delivery. We propose a modified form of Petri net where there are two colors of tokens, say red and green. Red tokens function precisely as in OCPN, and green tokens represent the arrival of PDUs on the network. For a transition to fire, it is be necessary not only for each of the places with incoming arcs to contain an unlocked red token, but it is also necessary for each place connected to the transition by outgoing arcs to contain a green token. We expect that such a semantics would allow the preservation of temporal relations such as *starts*, which is not possible in the current explicit release semantics except in a perfect network. On the other hand, such an extension to PO/PR service would constitute a major shift; for example, it is unclear how the POCv2 stream abstraction would integrate with such a semantics of delivery.

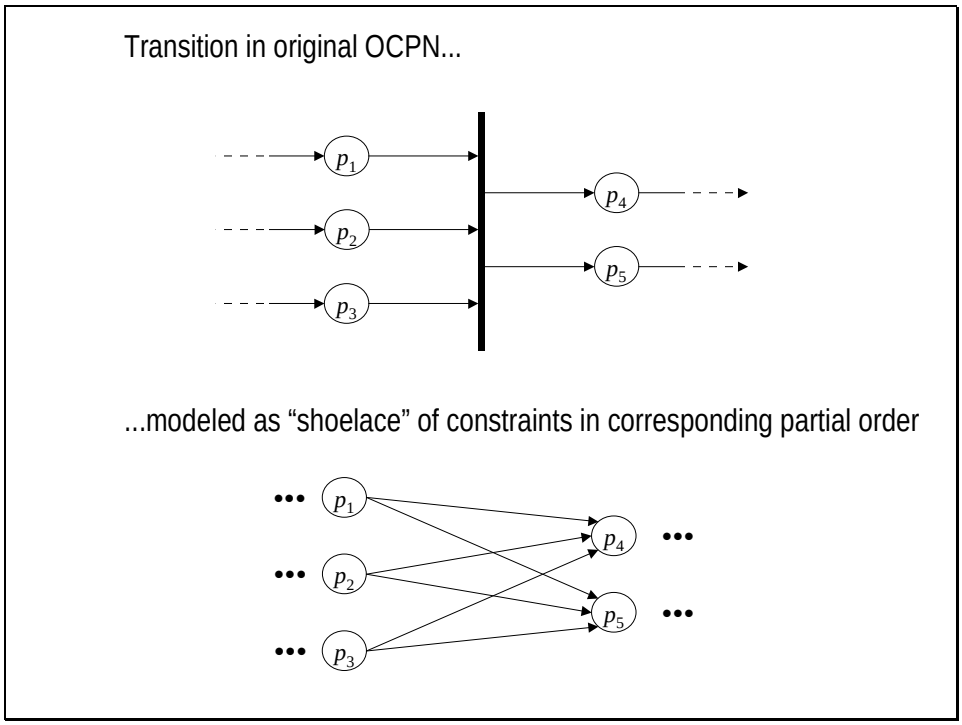

Figure 2.14 Example transformation from Lemma 2.1

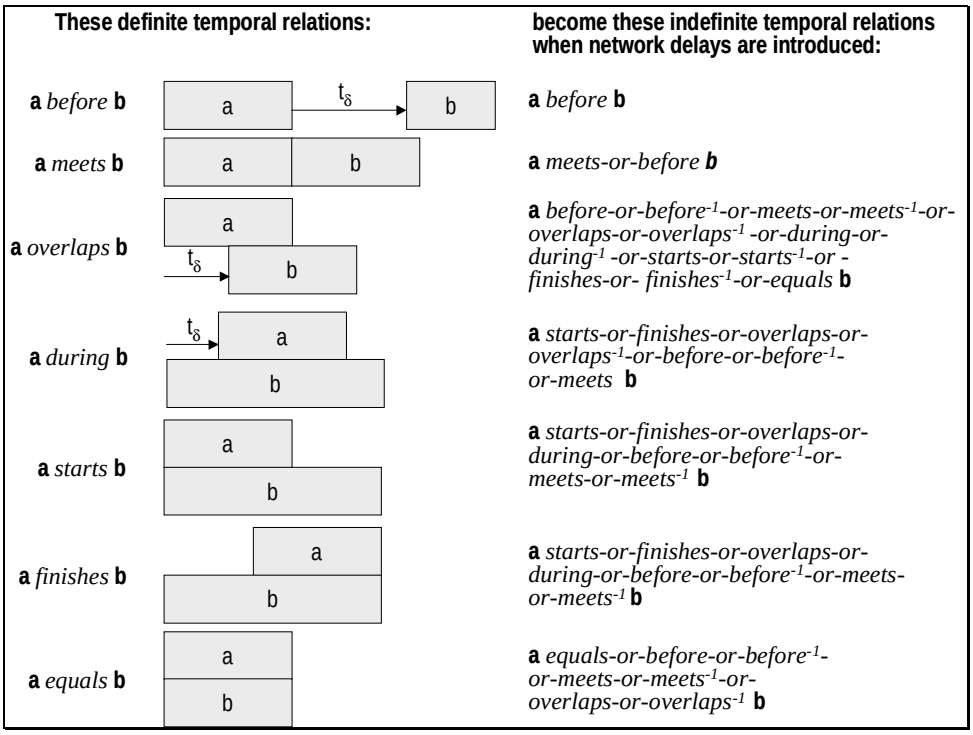

Figure 2.15 Indefinite intervals as a result of network delays

2.6.5 Objections to explicit release (and, motivation of data preview)

There is an objection to the explicit release feature. Consider again the example of a multimedia document in Figure 2.13 in which an image immediately follows an audio clip in the partial order. If the transport layer does not make data for successor objects available until after their predecessors are finished playing, the application may miss an opportunity to do some presentation layer processing on these objects. Here, we refer to presentation layer processing in the broadest sense, as any processing that is necessary to decode the transfer syntax of the APDUs, including, for example, decompression, parsing of a command, or decoding of image or audio data. The ALF principle suggests that presentation layer processing is likely to be a

bottleneck for such applications, thus anything that causes this processing to starve for data is to be avoided. (Clark and Tennenhouse, 1990)

How is this problem to be addressed? Most ALF-inspired work suggests that the transport layer simply provide only unordered service, and put all the burden of proper packet sequencing and/or synchronization on the application. In this dissertation, the author proposes a different approach that tries to provide the best of both worlds. We call this feature the *data preview*, or *buffer access* feature. With this feature, the transport layer still provides ordering (and with explicit release, synchronization) services. That is, it enforces the required delivery order. However, it also allows the application to do presentation layer processing out-of-order if and when idle time exists.

2.7 Data preview (buffer access) for support of integrated layer processing

One of the key architectural principles set forth in (Clark and Tennenhouse, 1990) is the notion that inessential sequencing constraints on protocol operations should be eliminated or at least reduced. For example, if it is more efficient to perform a particular data format conversion (a presentation layer function) before resequencing out-of-order messages (a transport layer function), the architecture should not create barriers that make this awkward. The traditional layered architecture does not permit such a resequencing of protocol operations: data cannot reach the presentation layer unless that transport layer completely relinquishes any ability to further process that data. What is desirable is a means by which resequencing can be done by the transport layer, and data format conversion can be done by the presentation layer, and these two functions can be done in either order. That is, depending on what is most convenient at a particular moment, the transport

and application entities can do either resequencing first, and presentation layer conversion second, or presentation layer conversion first, and resequencing second.

(Clark and Tennenhouse, 1990) introduce the term *Integrated Layer Processing (ILP)* to describe architectures that eliminate or reduce inessential sequencing constraints on protocol operations in multiple layers. Most approaches to ILP focus on integrating operations on a fine-grained level: for example, combining the checksum and data copy functions into a single loop so as to avoid touching each byte of data more than once. To distinguish this low-level approach from the approach which we introduce in this dissertation, we suggest the term *fine-grained ILP*. Fine-grained ILP operates at the level of bytes and instructions to perform as many operations as possible while each byte of data is in main memory. Fine-grained ILP has been studied, for example, in (Braun and Diot, 1995).

By contrast, we focus on a way to provide ILP at a message level. We call this *coarse-grained ILP* because it focuses on eliminating or reducing inessential ordering constraints among operations on entire messages. In particular, we propose a feature called *data preview*[18] to provide application, presentation and session layer functions to access and operate on messages that may still be buffered in the transport layer due to delivery order constraints.

The idea of data preview is simple; data preview provides the application layer access to data that is buffered in the transport layer due to delivery order requirements. Without a data preview feature, all processing on incoming ADUs must take place after data has crossed the transport service access point (TSAP)

[18] Programmers Note: Inside the UTL source code (in comments and variables names), the *data preview* function goes by the name *buffer access*.

boundary. In this situation, for an application is to take advantage of the benefits of out-of-sequence processing for presentation layer conversions, it must use an unordered transport service. This then implies that if there is any sequencing requirement in the *subsequent* processing of data—for example, ordering audio frames or video frames for playback, or providing a partial ordering or coarse-grained synchronization of document elements—such processing must necessarily be implemented in the application layer for *each* application, because after data delivery, the transport layer is unable to assist.

By contrast, with data preview, the application architect can consider two kinds of operations on data: (1) those that can take place on ADUs independent of sequencing (e.g., presentation layer conversions), and (2) those for which a total or partial processing order must be respected. Then, each time the application has time to process incoming data, it can ask the transport layer two questions instead of one. In the old architecture, the application simply asks: "Are there messages ready for delivery?". If the answer is "yes", the application can ask for delivery of these message(s) by issuing `Read()`requests. With data preview, the application may also ask: "Are there any messages buffered that have not yet been previewed?" If the answer to this second question is "yes", the application can request preview of these messages by issuing `Preview()` requests.

A preview request provides a pointer to the data, and the length of the data at that pointer. It is understood that the application is not allowed to modify any data outside this region. If the transport service is provided by the kernel, it is feasible that virtual memory mechanisms might be used to enforce this restriction. However, for the convenience of the application, a single read/write `(void *)` pointer value

(initialized to null) is also associated with the message. This value, called the application data pointer (`appDataPtr`) allows the application to associate some data with the ADU prior to delivery, and can be used in any way that the application designer sees fit. The transport layer does nothing with the value other than initializing it to NULL, and providing the application layer access to set and get its value. Some suggested uses for this pointer are as follows:

- The `appDataPtr` can be used as a pointer to an object or structure containing converted data for a presentation layer conversion where the local syntax consumes more space than is available inside the ADUs data area (i.e., the transfer syntax). The application may allocate an object or structure that will contain the converted data, and set the `appDataPtr` to point to this new object or structure. When the data is ultimately delivered, the application can check this pointer; if the `appDataPtr` is NULL, the application knows that the presentation layer conversion has *not* yet been done. If the `appDataPtr` is non-NULL, the application knows that the conversion *has* already been done, and the converted data is available.

- The `appDataPtr` can be treated as a Boolean flag indicating whether presentation layer conversion has been done or not. For example, for a presentation layer conversion that can be done in place (e.g., converting from big-endian to little-endian byte order), the value NULL can represent false, i.e., the presentation layer conversion has not been done, and the ADU is in network byte order, and 1 can represent true, i.e., the presentation layer conversion has been done and the ADU is in host byte order.

Data preview was incorporated into the design and architecture of UTL; but has not been fully implemented or evaluated; a complete evaluation of data preview is a subject for future work.

2.8 The relationship between partial order and partial reliability

The research on POC has been quite varied in terms of the relationship between partial order and partial reliability. In this section, we trace the history of this relationship in the research on POC, showing that the trend has been towards a separation of the partial order and partial reliability features. We then describe one innovation that actually proposes to move towards a tighter integration of the two—indeed, an integration between order, reliability and synchronization: namely the PR reliability class. This reliability class is designed to use the partial order and synchronization information to detect and take advantage of opportunities for extra retransmissions that would not ordinarily be feasible.

2.8.1 Previous work

Early papers such as (Amer et al., 1994) and (RFC1693) treat partial order and partial reliability as two equally important aspects of a single unified protocol, while still maintaining a separation between the semantics. That is, while partial order and partial reliability are considered together in nearly all discussions of POC, determination of whether or note a packet was to be declared lost was entirely orthogonal to considerations of delivery order. In the early designs and specifications of POC (for example, in the Estelle specification included in (Amer et al., 1994), the determination as to whether or not a packet should be declared lost was made by the application. The application provided the transport layer with a callback predicate function, `isObjectStillUseful()`. The transport layer would simply call this function any time it needed to make a determination as to whether to persist in trying to recover from the loss of a certain packet.

In later stages of the project, the trend has been to separate partial order from partial reliability. For example the analytic and simulation work of Marasli considers them separately, looking first at PO/R service, and then at U/PR service. Treatment of partial order and partial reliability in combination was deferred by Marasli to future work. At the earliest stages of implementation for POCv2, partial order and partial reliability were split into an upper ordering sublayer, and a lower reliability sublayer. As time progressed, the split become larger as the usefulness of an unordered reliable and unordered/partial reliable service become more apparent—particularly for the NETCICATS work.

The architecture that has emerged, with a clear split between the partial order and partial reliability functions, has certain advantages and certain disadvantages both from a protocol and an implementation perspective. However, we will defer a further discussion of those issues to Chapter 3, in the context of discussing the design and implementation of UTL. Instead, we turn now to a discussion of the PR reliability class.

2.8.2 The PR reliability class

POC introduced the notion of *reliability classes*. The reliability classes R for reliable objects and U for unreliable objects are motivated by the fact that some objects are absolutely essential to document content, while others are nice to have but strictly unnecessary. (Amer et al., 1994) proposes that there is a third useful reliability class, called PR, for objects that have usefulness during some period of time, but which later become useless. The example given in (Amer et al., 1994) is a caption or subtitle on a movie, where the caption is useful during the scene with which it corresponds, but after the dialogue has moved on, the caption becomes useless;

presenting it late would only be distracting, and it is not important enough to halt the progress of the document. The concept of the PR reliability class is that the transport layer will try to retransmit PR objects for some limited period of time, and then will *declare them lost.*

POCv2 retains the notion of the PR reliability class. However, on three crucial design decisions, POCv2 diverges from POC. These design decisions are:

- What component in the architecture makes the determination that an object should be declared lost?
- How is the determination made?
- When it is made?

In the case of POC, the answers to these questions are as follows. The POC receiver polls the application receiver (on the basis of a timer) to inquire about the status of each outstanding object in the partial order. The application makes a callback function `isObjectStillUseful(objectNum)` available to the transport layer. Each time this function is called, the application must make a determination as to whether the given object is still useful, or whether it should inform the transport layer to give up on this object.

For POCv2, we propose an entirely different way of making this determination. We base our proposal on the following observations.

First, reviewing the usefulness of *every* outstanding object in the partial order is wasteful. The only objects that should be considered as candidates for being declared lost are those objects which, if they were declared lost, would allow for the earlier delivery of some other object. The transport layer has enough information to determine which objects these are.

For example, consider the example partial order from Figure 2.16. For sake of argument, assume that all objects are in the PR reliability class. The figure shows that at some point in the connection, objects 0,1 and 3 have been received, but object 2 has not been—presumably, it was lost, and will have to be retransmitted. At this point, it is reasonable to consider declaring 2 lost. However, it would be pointless to consider the "losability" of objects 4 and 5. Since these objects cannot be delivered until after object 3 is delivered, there is no reason to consider declaring 4 or 5 lost until object 3 has *been* delivered. Furthermore, *even then* there is no reason to consider declaring 4 or 5 lost unless and until some *later* object in the partial order (say, object 6) arrives at the receiver. Then, and only then, would it be helpful in some way to consider sacrificing the delivery of 4 and/or 5 for the sake of earlier delivery of object 6.

Second, even if the transport layer were to make a determination as to which objects are reasonable candidates for review of their losability, and call the `isObjectStillUseful()` function on these objects only, it is unreasonable to declare an object (say, object 2 in Figure 2.16) lost until the moment that the application actually requests some data via a `Read()` operation. Note that the application may not be in a constant state of consuming data. The structure of many applications is to read some data from the transport layer, and then perform some processing on that data before requesting additional data. During that time, additional objects may show up at the transport layer. Declaring them lost prematurely would be counterproductive.

Thus, rather than periodically reviewing the usefulness of every outstanding PR object, POCv2 instead provides the following semantics for PR

delivery. Note that while the reliability classes are defined at the object level, the operation of declaring data lost occurs at the cell level in POCv2:

Definition of PR reliability in POCv2

> A POCv2 PR cell is *declared lost* when and only when all three of the following conditions become true:
>
> (1) no data is currently deliverable without declaring one or more PR cells lost,
>
> (2) some data will become deliverable if a PR cell is declared lost, and
>
> (3) the application has requested data via a pending `Read()` operation.

Thus, in POCv2, cells of objects with the PR reliability class are declared lost as a side-effect of the `Read()` operation, and the transport receiver is the sole determiner of when to declare a cell lost. Chapter 6 describes the algorithm and data structures used by the POCv2 receiver to make this determination efficiently.

From the perspective of the sender, PR cells are treated as reliable by the sender for purposes of retransmission, except that the POCv2 receiver may send a special acknowledgment indicating that the object has been declared lost. This ack cancels any pending retransmission of these cells. While the usual semantics of an ack is “object successfully received”, an acknowledgment for a PR object has the semantics “no further transmission needed”.

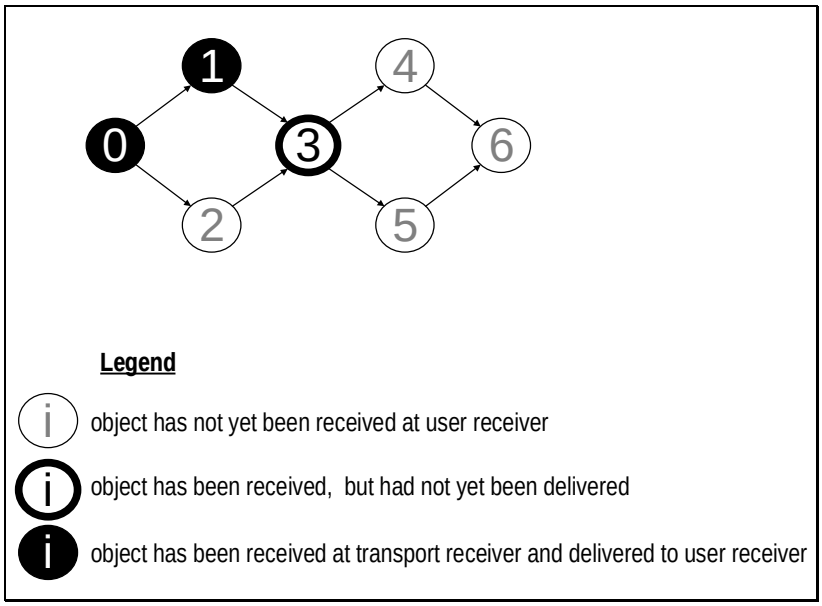

Figure 2.16 Example PO for explanation of PR reliability class

2.8.3 PR reliability + explicit release synchronization = graceful degradation

In this section, we describe how the combination of PR reliability and explicit release synchronization can provide for graceful degradation of multimedia documents when the network loses or reorders packets. We first describe a phenomenon we refer to as *slack time* in the playout of a multimedia document. We then describe how the PR reliability class and explicit release can be used to take advantage of slack time to allow extra retransmissions (and therefore, increased reliability) with no increase in delay.

Slack time occurs in the presentation of a multimedia document when the playout of a document reaches a point where there is nothing for the application to do but wait for some packet to become deliverable. Slack time may occur in a document

for several reasons. The most obvious is when explicit pauses are included in the document. For example, consider a portion of a document consisting of:

- a single image, accompanied by some text (perhaps a riddle or joke)
- followed by an explicit pause of 5 seconds,
- followed by a piece of text (the riddle's answer or the joke's punch line).

As soon as the document browser has retrieved and displayed the image, the browser releases the successor of that image, which is the pause element. The pause element does not release its successor, the answer or punch line, for 5 seconds. During this time, the browser has absolutely nothing to do.

The key idea is that during this time, the transport layer can be accumulating packets in the buffer that may follow the punch line in the partial order, so that when playout of the document proceeds, a substantial portion of the document has already been received, and is waiting in the transport layer's buffers. If the underlying network is experiencing high loss, this extra pause time can allow for more retransmissions. Because the pause is a natural part of the document, these extra retransmissions add no delay to the playout of the document as a whole.

Explicit pauses generally have a specific length. More interesting are document elements resulting in slack time that may be unpredictable in length, since this is where we can take advantage of the PR reliability class. The idea is that we specify the PR reliability class for objects that we might otherwise treat as unreliable (audio, for example). Then, if the slack time allows for retransmission of missing cells, we gain in terms of reliability. If it does not, we cancel the retransmissions; the only price paid is the processing cost of declaring the cells lost, and bandwidth cost of

sending the acknowledgments canceling the retransmissions. In Chapter 6 we establish that this processing cost is reasonable. As for the acknowledgments, we observe that algorithms that provide for TCP-friendly congestion avoidance may send acknowledgments (or negative acknowledgements) for best-effort traffic anyway, even if no retransmission takes place. (see, for example, Jacobs and Eleftheriadis, 1997).

Both *continue buttons* and audio elements can produce varying amounts of slack time. A continue button is an area in the document that the user must click to allow the document to proceed. A continue button provides for a pause of indefinite length. Another less obvious cause of slack time is the draining of audio buffers. To create smoother audio playout, the application may build up a small playout buffer of audio data before it begins playing the audio samples through the output device; this enables the application to compensate for jitter. Many audio devices also have their own internal buffers, either on the audio hardware, or in the device drivers. Thus, it is reasonable to expect that the application may have some slack time at the end of a particular audio clip where it is simply waiting for the audio device to drain the remaining audio samples before continuing. This slack time provides another opportunity for extra retransmissions of the portions of the document that follow the audio clip.

2.8.4 Unresolved issues and future work for the PR reliability class

The central focus of this dissertation is a performance evaluation of PO/R service. Thus, the implementation of the U and PR reliability classes in POCv2 is incomplete. Future work will include completion of an implementation of the U and PR reliability classes, and an evaluation of the tradeoffs between reliability and delay. Of particular interest will be evaluating the extent to which pauses in a multimedia

document can allow for extra retransmissions of PR objects—particularly audio—as described in Section 2.8.3.

One area for future development is the notion of separate reliability classes for individual cells. For example, it may be useful to consider an MPEG video clip to be a single object in the partial order, but to be able to define separate reliability classes for I, B and P frames. While the basic algorithm could still be used, the challenge would be to efficiently transmit to the POCv2 receiver the reliability classes of individual cells in a stream. If there is a specific periodic repeating pattern (as in MPEG[4]), this would be fairly straightforward. Arbitrary reliability vectors at the cell level would present a more difficult challenge in terms of balancing the overhead of sending the reliability vector vs. the benefit of providing PR delivery.

2.9 Providing control over reliability via ADN-cancel

The *ADN-cancel* feature allows cancellation of messages that have already been submitted to the transport layer. The application specifies an Application Data Name (ADN) for each message and can request that the transport layer cancel the 3transmission of any message (or group of messages) by specifying its (their) ADN. To motivate this feature, consider a system for transmitting images, where a human or automated decision system has to make a critical decision based on the image received, such as:

- in a telemedicine application, whether or not a patient should be transported to a field hospital.
- in a battlefield situational awareness application, whether a given image represents friend or foe, non-target vs. target.

At some point, the human receiving the image may determine that she/he has received sufficient data to make a decision. At this point, the receiver would like to cancel the transmission of all additional packets carrying an ADN for the image in question, without severing the connection. Keeping unnecessary traffic off the air may be particularly useful in situations where multi-access communication makes bandwidth a scarce commodity, or where radio transmissions may help an enemy locate a target.

While a facility for canceling messages may seem like an obvious transport service feature, surprisingly it is not available in TCP. True, the receiving application can signal to the sending application that no more data should be submitted to the transport layer for a given image. However, neither the receiving application nor the sending application can do anything to cancel the sending of data that is already in the transport layer "pipe", short of aborting the connection.

The closest one could come with TCP is to use the urgent data feature to mark the position in the byte stream where the data for the cancelled image ends. This is what is done, for example, in FTP or Telnet, when the flow of data is interrupted (for example, by CTRL/C.) The receiving application is then notified immediately of the position in the data stream where the cancelled image ends. Because this notification occurs immediately, it is sometimes called "out-of-band data", however this is a misnomer. The only thing sent out-of-band is the sequence number in the data stream where the cancelled image ends. The sending and receiving TCP entities still must process every byte of this now unnecessary data, and the receiving application must read every byte of it as well. The ADN-cancel feature adds

the ability for the sending transport entity to simply flush the now unnecessary packets out of the transport layer altogether.

ADN-cancel is implemented in version 0.90 of UTL, however it has not been extensively tested or evaluated in practice. In particular, it has not been incorporated into any applications other than a brief test in the regression routines described in Section 3.7.2. Future work will include the incorporation of ADN-cancel into the ReMDoR browser, and evaluation of this feature.

One of the key difficulties with implementing ADN-cancel is the overhead of maintaining a dictionary of ADN names. It is crucial that the add, delete, and find operations in this dictionary be efficient. The current version uses the `tsearch` routines that are part of the Unix system library. According to the Unix `man` page, these routines are based on the binary tree search from "Knuth (6.2.2) Algorithms T and D." Hashing might be a preferable alternative, but because it is difficult to predict how ADN-cancel might be used in practice (in particular, what programmers might choose to use as the ADN values), it is difficult to know what the distribution of keys might be. Evaluating each of these approaches is another topic for future work.

The ADN-cancel feature has currently been implemented only for unordered/partially-reliable (U/PR) service. Future work includes considering how this feature can be integrated into a partial order/partial reliability service. Along the same lines, it may be useful to add features into POCv2 to allow canceling the remainder of a period or epoch, particularly if a period or epoch corresponds to a particular multimedia document, or portion thereof. For example, one might imagine a situation where clicking on a hyperlink would cancel the remainder of the current period, while setting up a new epoch with the service profile of the document

referenced by the hyperlink. On a philosophical note, the ability to cancel an entire class of messages or an entire activity (such as a period corresponding to a single document) has the flavor of a session-layer service or protocol. In spite of the fact that session layer functionality could be useful, TCP/IP lacks any standardized session layer functions. Recent work has proposed adding some session-like features into HTTP (Stevens, 1996), so this may an idea whose time has come (again).

2.10 Current status of POCv2 implementation and areas for future work

The design of POCv2 described in this chapter has been partially realized in the Universal Transport Library (UTL) described in Chapter 3. In particular, the following features of POCv2 are fully implemented in UTL. Each has been extensively tested and evaluated as part of the UTL regression testing described in Section 3.7.2, and the performance experiments in Chapter 5:

- partially-ordered /reliable[19] service, with periodic partial orders as in POC
- the stream abstraction
- provision for a default service profile for epoch 0, and negotiation of an alternate service profile for epoch 1
- explicit-release synchronization

The following features are included in the design of the data structures and API, but are either only partially implemented, or unimplemented in the current version; completion and evaluation of these features constitutes an area for future work:

[19] As an interim measure, all objects are currently treated as if they were assigned reliability class R, regardless of the reliability class passed in the service profile.

- partial order/partially reliable service, including the U and PR reliability classes.
- negotiation of the service profile for epochs beyond epoch 1 (implemented, but not fully tested or debugged.)
- the data preview feature (provided for in the API, but unimplemented)
- ADN-cancel for partially-ordered/partially-reliable service (ADN-cancel is implemented for unordered service.)

2.11 Chapter summary

We described the innovations in transport layer service developed and investigated by the author as part of this dissertation. Specifically, we described key ways in which the author's protocol, Partial Order Connection version 2 (POCv2) differs from the Partial Order Connection (POC) specified in (Amer et al., 1994). The main theme underlying these new features is to make POC a useful protocol for multimedia applications, which has been the key example motivating PO/PR transport service since first proposed in (Amer et al., 1994). The innovations includes features for more natural and efficient specification of the partial order and reliability requirements of the document, provision for multiple partial orders (enabling persistent connections), and integration of coarse-grained synchronization support.

We also propose two additional features that have broader applicability to transport protocols in general:

(1) a data preview feature enabling coarse-grained integrated layer processing, which can apply to any protocol providing ordered or partially-ordered service, and

(2) an ADN-cancel feature which can be applied to any protocol providing reliability.

In the next chapter (Chapter 3), we describe the Universal Transport Library: a software system designed for rapid prototyping and performance evaluation of transport layer protocol innovations such as the ones just presented.