Partial-Order Transport Service for Multimedia and Other Applications

Paul D. Amer, Christophe Chassot, Thomas J. Connolly, *Student Member, IEEE,* Michel Diaz, *Senior Member, IEEE,* and Phillip Conrad, *Student Member, IEEE*

Abstract— **This paper investigates a partial-order connection (POC) service/protocol. Unlike classic transport services that deliver objects either in the exact order transmitted or according to no particular order, POC provides a partial-order service; that is, a service that requires some, but not all objects to be received in the order transmitted. Two versions of POC are proposed: reliable, which requires that all transmitted objects are eventually delivered, and unreliable, which permits the service to lose a subset of the objects. In the unreliable version, objects are more finely categorized into one of three reliability classes depending on their temporal value. Two metrics based on $e_i(P)$, the number of linear extensions of partial-order P in the presence of i lost objects, are proposed as complexity measures of different combinations of partial order and reliability. Formulae for calculating $e_i(P)$ are derived when P is series-parallel. A formal specification of a POC protocol, written in Estelle, is presented and discussed. This specification was designed and validated using formal description tools and will provide a basis for future implementations.**

I. Introduction and Motivation

CURRENT applications that need to communicate objects (i.e., images, files, sound bites) choose between classic transport services that provide either an ordered service (e.g., TCP) or one that does not guarantee any ordering (e.g., UDP). Many applications, however, such as multimedia only require partial-order delivery; some objects being communicated must arrive in the order transmitted, some may arrive in any order. By currently using an ordered service, these applications waste both memory and bandwidth resources and at the same time risk incurring greater delays.

Multimedia traffic often is characterized either by periodic, synchronized parallel streams of continuous bit rate (CBR) information (e.g., combined audio-video), or by structured image streams (e.g., displays of multiple overlapping and nonoverlapping windows). Currently these applications must use and pay for an ordered service even though they do not need it. Because a partial-order service has greater flexibility in delivering objects to a user, such a service will reduce delays in object delivery, and require less memory and/or bandwidth on the average than would an ordered one. This will be the case when the underlying service is inherently unreliable as in the Internet packet switched network. In today's age of megabyte objects, avoiding the need to buffer or retransmit even one object can result in significant savings.

Manuscript received January 23, 1993; revised July 7, 1994; approved by IEEE/ACM Transactions on Networking Editor K. Sabnani. This work supported by CNET under Grant 92 1B 178 as part of a CNET-CNRS collaborative project on High-Speed Multimedia Systems. P. D. Amer, T. J. Connolly, and P. Conrad were supported in part by the NSF under Grant NCR-9314056, the U.S. Army CECOM, Ft. Monmouth, and the U.S. Army Research Office under Contract DAAL03-91-G-0086, DAAL03-92-G-0070.

P. D. Amer, T. J. Connolly, and P. Conrad are with the Department of Computer and Information Science, University of Delaware, Newark, DE 19716 USA (email: amer@cis.udel.edu).

C. Chassot and M. Diaz are with the Laboratoire d'Automatique et d'Analyse des Systemes, Centre National de la Recherche Scientifique (LAAS du CNRS), Toulouse, France (email: diaz@droopy.laas.fr).

IEEE Log Number 9406181.

Two variations of a partial-order service are proposed: reliable partial-order service (R-PO) which guarantees the eventual delivery of *all* transmitted objects according to a defined partial-order, and unreliable partial-order (U-PO) service which makes a best effort to deliver all transmitted objects, but tolerates a well defined level of lost objects. In addition to introducing partial-order services/protocols, this article considers quantifying, comparing and formally specifying each version.

Additionally, this article investigates metrics that characterize (i.e., quantify) the work that must be performed to provide a particular R-PO (U-PO) service, and how this metric is computed for a given partial-order. Such a metric would permit one to compare two or more R-POs (U-POs) thereby distinguishing between different quality of service levels and providing a clearer means, say, for charging for each service.

Also, a U-PO service allows a destination to presume certain, but not all objects to be lost when their temporal value has expired. This article considers: how one classifies objects according to their varying temporal constraints; how a destination dynamically decides when objects are lost; and what are the effects of such a decision.

These issues are considered as follows. Section II introduces and motivates R-PO and U-PO service in detail. Section III proposes two metrics based on a partial order's set of *linear extensions* for quantifying and comparing the complexity of different R-PO and U-PO services. Formulae for calculating these metrics are derived for the subset of partial orders that are series-parallel, a form often appropriate for characterizing multimedia applications. In Section IV and Appendix A, a partial-order connection (POC) protocol for providing either R-PO or U-PO service for any partial order is specified in the ISO formal description technique, Estelle. Also several practical issues of concern in implementing POC in the future are considered. Finally, Section V provides conclusions and directions for future research.

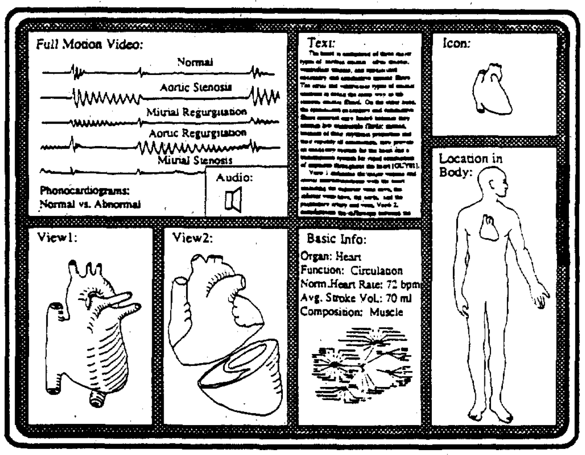


Fig. 1. Anatomy and physiology instructor workstation instance (Example 1).

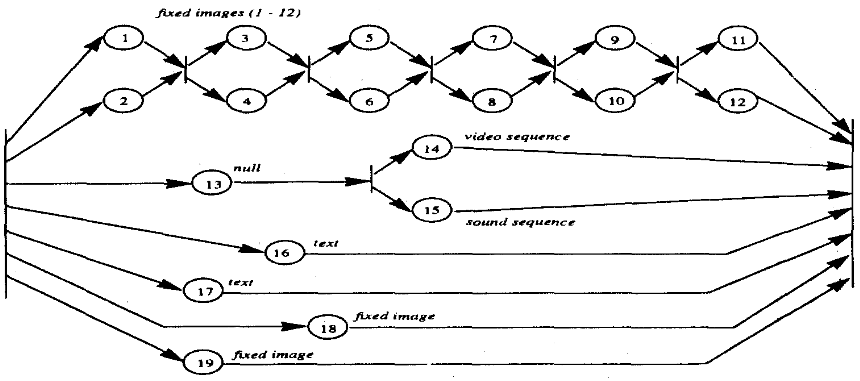


Fig. 2. Analogous partial order (Example 1).

II. Partial-Order Service

Partial-order services are needed and can be employed as soon as a complete ordering is not mandatory. When two objects can be delivered to a transport service user in either order, there is no need to use an ordered service that delays delivery of the second one transmitted until the first arrives. Four illustrating examples are presented.

Example 1: Consider an Anatomy and Physiology Instructor system described as "a simple multimedia application example based on the hypermedia paradigm and temporal relation specification [22]." Here a workstation displays multiple windows of video, audio, text, image, and animated image according to well defined synchronization and ordering requirements. In one particular presentation, the user learns of the human heart by combining an animated image and sound track of a heart pumping in one window while simultaneously providing general textual information (e.g., average heart rate) in another window (see Fig. 1 taken from [15].)

Fig. 1 adapted from [22] illustrates the partial order that models the heart presentation. Views 1 and 2 of the heart are fixed images that change with time to provide a gradual rotation or slow-motion animation. These images are represented by objects 1-12 displayed in a sequence of six pairs. There are two text objects and two single image objects shown in the four windows on the right. These are single independent objects (16-19). The full motion video and sound track are represented here as single objects preceded by a null object used for synchronization purposes (13-15).

The six pairs of images of the slow motion animation have an inherent order, but there is no order constraint on the delivery within each pair. Similarly the arrival ordering of all twelve images is independent on the ordering of the four single objects. That is, the arrival of some objects is only *partially dependent* on all of the others.

Note that a partial-order service focuses primarily on delivery order. The temporal value of objects is taken into consideration when the service allows some level of permitted loss (Section IV–B). It is assumed that synchronization concerns in presenting the objects after delivery is a service provided on top of the proposed partial-order service. Temporal ordering for synchronized playback is considered, for example, in [3], [16].

Example 2: Consider an application that must do a screen refresh on a workstation screen/display containing multiple windows. In refreshing the screen from a remote source, objects (icons, still or video images) that overlap one another

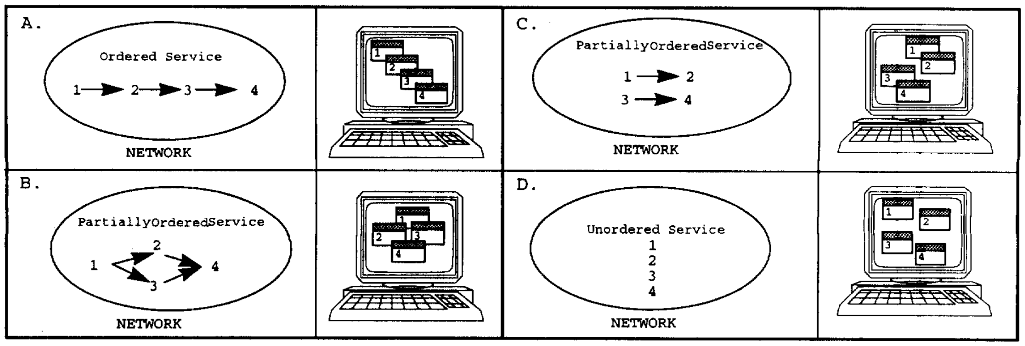


Fig. 3. Ordered versus partial ordered versus unordered service (Example 2).

Fig. 4. Television news broadcast for the hearing impaired (Example 3).

have a "series" relationship and should be delivered to the display for refreshing from bottom to top for optimal redisplay efficiency. However, objects that do not overlap have a "parallel" relationship and may be refreshed in any order. Therefore, the way in which the windows overlap induces a partial order.

Consider the four cases in Fig. 3(a) sender wishes to refresh a remote display that contains four active windows (objects) named {1 2 3 4}. Assume the windows are transmitted in numerical order and the receiving application refreshes windows as soon as the transport service delivers them. If the windows are configured as in Fig. 3(a), an ordered service, also referred to as a FIFO channel, is required. In this case, only one ordering is permitted at the destination. If window 2 is received before window 1, the transport service cannot deliver it or an incorrect image will be displayed.

At the other extreme, if the windows are configured as in Fig. 3(d), an unordered service would suffice. Here any of 4! = 24 delivery orderings would satisfy the application since the four windows can be refreshed in any order. As notation, four ordered objects are written 1;2;3;4 and unordered objects are written using a parallel operator: 1 || 2 || 3 || 4.

Fig. 3(b) and (c) demonstrate two (of many) window configurations that call for a partial-order delivery service. In these cases, two and six orderings, respectively, are permitted at the destination.

Example 3: Sending a set of objects in a partial order need not be a one time event as in the previous two examples. In cases of periodic (i.e., cyclic) communication such as a multimedia presentation with synchronized video, sound and text streams, a partial order models each of a repeating pattern of objects. In this case, each repetition or *period* represents a single partial-order snapshot in a stream of sequential periods of communication. This example and the one that follows both illustrate periodic communications.

Consider a television news broadcast for the hearing impaired as shown in Fig. 4. This multimedia broadcast includes two video components (normal broadcast and sign language broadcast), two audio components (left and right channels), and one text component (the subtitle). Assume the five components have differing characteristics so that in each second there are 30 images/s for the main video component, 10 images/s for the hand signing, 60 sound fragments/s for each audio channel, and one subtitle text object per second consisting of either 1) new text, 2) a command to repeat the previous second's text, or 3) a command for no text. These 161 objects are repeated each and every second for the entire news transmission.

Fig. 5's partial-order models the delivery the characteristics of the multimedia presentation. Objects connected by a horizontal line must be received in left to right order, while those in parallel have no inherent ordering requirement. While objects within each of the given five streams must be received in order, there exists flexibility in delivering objects in different streams.

Example 4: This final example illustrates how performance can be improved with a partial-order service. Improvements are expected in several areas including: memory utilization, delay, throughput, and bandwidth utilization. The following hypothetical example highlights these gains. The values and calculations below are not intended to be a rigorous analysis; their purpose is simply to provide a more concrete illustration of the potential savings.

Hypothesis: Assume an ATM network running on a 150 Mbps channel. Assume that 44 of the 53 octet cells are available for user data thus resulting in an actual available

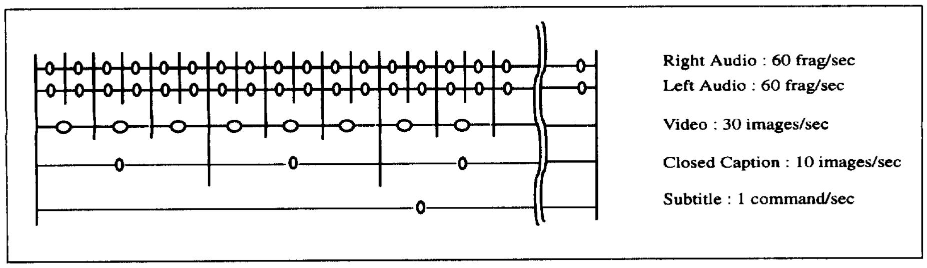


Fig. 5. Analogous partial order (Example 3).

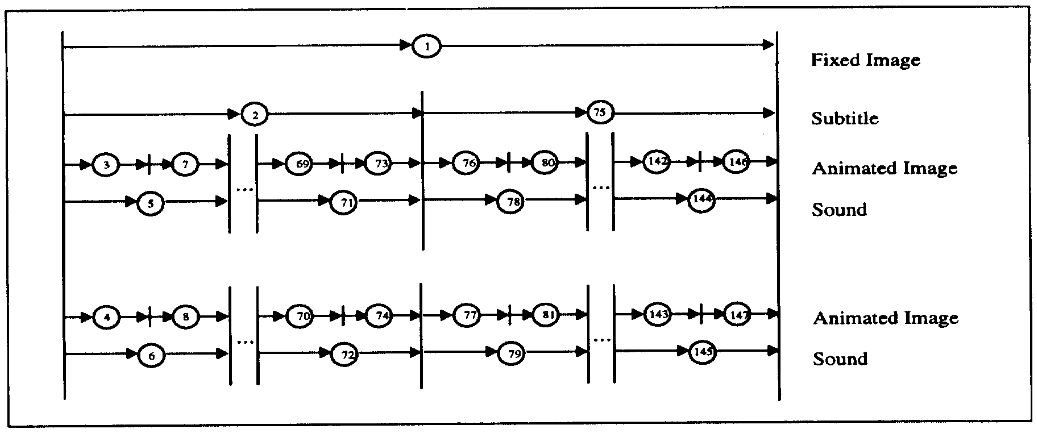


Fig. 6. Hypothetical multimedia example for performance analysis (Example 4).

bandwidth of roughly 125 Mbps. The following table summarizes the size and transmission time of different objects:

object type	size (octets)	transmission time (ms)
animated image (ai)	50,000	3.200
sound fragment	320	0.020
subtitle	500	0.032
fixed image	30,000	1.920

Assume a roundtrip time (including delays at intermediate packet switches) of 200 ms and a multimedia application using the ATM network to remotely present two animated image sequences, both with sound and one with an ongoing subtitle; and one presentation with fixed images. These three presentations are spatially and temporally independent. A two-second period representing the repeating partial-order service needed is shown in Fig. 6.

Assume the sender sends objects in numerical order and only object 1 is lost or damaged by the underlying network service. At the receiver depending on its receive window size, a classic ordered transport protocol will buffer or reject arriving objects 2 through k where k is the last object received before a retransmission of object 1 arrives. To the contrary, a partial-order transport receiver will accept and immediately deliver objects 2 through k to the user since their delivery is independent of object 1. No buffering is required.

Suppose the strategy is to buffer the out-of-sequence objects and to use selective positive acknowledgments with a sending retransmission timer value of 250 ms.

Case 1) No Flow Control: Before the 250 ms timeout, the sender will send objects 2 through 119 (taking into account the transmission time of each object), that is, 2 subtitles, 38 sound fragments and 78 animated images. For an ordered service, the needed buffer space for these objects will be 3.7 Mbytes. For a partial-ordered service, no buffers are needed. Additionally, the average end-to-end delay for objects 1–119 will be 231.1 ms for an ordered service while only 104.2 ms when a partial order is used. In this scenario, the advantage of a partial order is clear.

Case 2) with Flow Control: Assuming real-time replay of the information at the receiver, one may assume that the transmission of any animated image stream will be throttled to an approximate rate of 25 images/s or 1 every 40 ms. In this case, the sender will output a maximum of 7 images in each image stream before retransmitting object 1. That is, only objects 1-24 will be outstanding. Just prior to receiving the retransmission of object 1, a classic ordered transport protocol receiver will need to buffer 687 Kbytes, again an amount of memory not needed with a partial-order service. Similarly for objects 1-24, the average delay will be 238.4 ms versus only 112.4 ms for the partial-order service.

This example, while contrived, is meant to demonstrate that the potential quantitative gains in using a partial order are nontrivial. More detailed studies of actual expected gains are in progress and will be discussed further later on.

In summary, these four examples illustrate the usefulness of a partial-order service. They also illustrate that the partial order is dependent on the application and may be specified at different levels. Compare the partial order used in Example 1 where an entire video sequence is a single object to the cases in Examples 3 and 4 where individual video frames are single objects. The efficiency gained with a partial-order service will depend on how an application designer chooses an appropriate structure and granularity for the partial order.

A. Reliability Versus Order

While the most common transport protocols (e.g., TCP) work hard to avoid the loss of even a single object, most multimedia applications have a genuine ability to tolerate loss. Losing one frame per second in a thirty frame per second video, or losing a segment of its accompanying audio channel, is usually not a problem. Bearing this in mind, the proposed partial-order transport service combines partial order with varying levels of loss that can be tolerated. Different loss levels provide different levels of *partial reliability.*

Traditionally there exist four transport services: reliable-ordered, reliable-unordered, unreliable-ordered, and unreliable-unordered (see Fig. 7). Reliable-ordered service is denoted by a single point where all objects are delivered in the order transmitted. Traditional file transfer is an example application requiring such a service. Reliable-unordered is a single point where all objects must be delivered, but not necessarily according to the order transmitted. Transaction processing such as credit card verifications requires such a service.

Unreliable-ordered service allows some objects to be lost; those that are delivered, however, must arrive in relative order[1]. Since there are varying degrees of unreliability, this service is represented by a set of points in Fig. 5. An unreliable-ordered service is applicable to packet-voice or teleconferencing applications. If duplicates are not permitted, this represents what some, but not all authors call "at-most-once" delivery service [20].

Finally unreliable-unordered service allows objects to be lost and delivered in any order. This is the kind of service used for normal email (without acknowledgment receipts) and electronic announcements or junk email.

The concept of a partial order expands the order dimension from the two extremes of ordered and unordered to a range of discrete possibilities. R-PO service, for example, is appropriate for the screen refresh described earlier in Example 2. U-PO service is appropriate for general multimedia applications such as the television news broadcast for the hearing impaired in Example 3.

B. Related Work

Other authors have considered theoretical consequences of channel ordering, or lack thereof, in the context of designing

[1]An unreliable service (e.g., common mail delivery by the postal system) does not necessarily lose objects, simply it may do so without failing to provide its advertised quality of service.

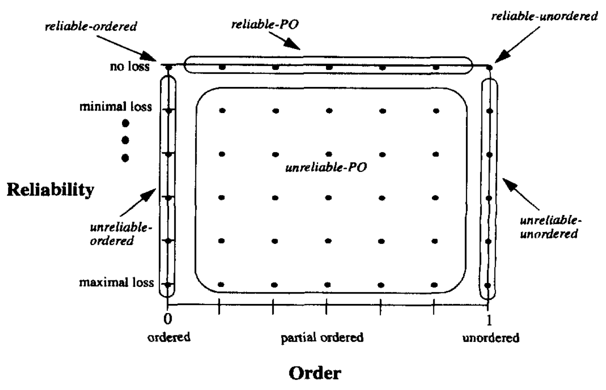


Fig. 7. Quality of Service: reliability versus order.

and verifying distributed algorithms [19], [23]. For example, Ahuja shows that some conclusions derived on the design of distributed algorithms need not have required FIFO ordering as a base assumption [4]. Ahuja, however, assumes a sending process dynamically builds the partial order and that no objects are ever lost [13], [26]. Our work assumes a predetermined partial order negotiated between the sender and receiver and U-PO service allows objects to be lost. Also, Ahuja's four data types do not permit all possible partial orders of objects as does Protocol POC.

Peterson *et al.* define a partial order on the messages communicated by a set of distributed processes and implements a protocol *Psync* that encodes the partial ordering within each message [24]. The partial order is defined by the interleaved times that messages are sent and received in the shared message space of the multiple communicating processes and dynamically changes with each newly sent message. Our work differs in its assumption of a point to point connection in which both sides agree at any point in time to a partial ordering of the data to be transferred.

III. Quantifying and Comparing Partial-Order Services

The complexity of a partial-order P can be quantified by its set of linear extensions, denoted $L(P)$. Each linear extension in the set $L(P)$ is essentially one of the orderings of the objects that is permitted at the destination. The number of linear extensions of P, denoted $e(P)$, is thought as the best single number which measures the complexity of P [28]. Clearly for N objects, e(complete order) = 1, e(no order) = $N!$

It is argued in [1] that $e(P)$ appropriately quantifies a desired partial order transport service in communication networks. Intuitively this metric correlates to the work a protocol would have to perform to provide a particular partial-order service. This is because the larger the number of permitted orderings allowed at the destination, the less overhead is expected to provide acceptable object delivery. For example, the larger the number of allowable orderings, the smaller the expected demand for memory to temporarily store objects received out of order as shown in Example 4 in Section II.

Several interesting questions related to linear extensions and therefore to a partial order service are now discussed.

A. Reliable Partial-Order Service

Given a particular partial order, just how many orderings are permitted at the destination if no losses are permitted? Answering this question allows one to quantify and compare two or more R-PO services. Unfortunately, there is currently no known formula for calculating $e(P)$ for an arbitrary partial order. Recently it has been shown that the problem of computing $e(P)$ is $\#P$-complete[2] [9]. There is an $O(N^5)$ algorithm, where N is the number of objects, for computing $e(P)$ for partial orders that form a tree when all edges are considered undirected [7]. Similarly, there is an $O(N^8)$ algorithm for computing $e(P)$ for any graph (and therefore for any partial order) where if the directions of the edges of P are not considered, any resulting cycles are edge disjoint [12]. Neither of these forms, however, model multimedia applications.

If the partial orders under consideration are series-parallel [28], calculation of $e(P)$ is possible [1]. While not all applications calling for a partial-order service need a series-parallel one, such a composition is reasonable for many applications, particularly multimedia applications. For instance, the partial orders in all of Section II's examples are series-parallel. (Similarly, the Object Composition Petri-nets proposed by Little and Ghafoor as a basis for modeling the synchronization and ordering of multimedia entities often are series-parallel [21], [22].)

Using ";" and "||" as notation for series and parallel composition, respectively, the Anatomy and Physiology Instructor multimedia presentation of the human heart (Fig. 1) can be defined as

$$((1 \parallel 2);(3 \parallel 4);(5 \parallel 6);(7 \parallel 8);(9 \parallel 10);(11 \parallel 12))$$

$$\parallel (13;(14 \parallel 15)) \parallel 16 \parallel 17 \parallel 18 \parallel 19$$

From [28, Example 3.5.4], the following formulae for series-parallel compositions are known. If $X_1, \ldots, X_k$ are k partial orders with $n_1, \cdots, n_k$ objects, respectively, then $X_1; \cdots; X_k$ and $X_1 \parallel \cdots \parallel X_k$ have $N = \sum_{i=1}^{k} n_i$ objects, and

$$e(X_1; \cdots; X_k) = \prod_{j=1}^{k} e(X_j)$$

$$e(X_1 \parallel \cdots \parallel X_k) = \frac{N!}{\prod_{j=1}^{k}(n_j)!} * \prod_{j=1}^{k} e(X_j).$$

Note that these formulae differ only by a multinomial coefficient which accounts for the allowed interleaving in parallel composition.

[2] $\#P$-complete is a similar concept to NP-complete, however, it refers to counting problems rather than decision problems. The significance is that it is unlikely that polynomial time algorithms exist for $\#P$-complete problems.

B. Unreliable Partial-Order Services

Just how much more flexible is a partial order with partial reliability (i.e., loss is permitted)? Suppose a destination application not only permits objects to arrive in a partial order, but that it also tolerates an occasional missing object. Let $e_i(P)$ denote the number of linear extensions permitted by a partial-order P that tolerates the loss of **exactly** i objects where $e_0(P)$ represents what previously was denoted $e(P)$. This section provides formulae for $e_i(P)$ analogous to those in the previous section whenever P is series-parallel.

If a receiving application can tolerate the loss of some objects, then the destination partial-order service conceivably could more flexibly deliver those objects that arrive out of order from the network (even in terms of a defined partial order) by simply assuming certain expected ones were lost. The amount of added flexibility when i objects can be lost can be quantified by considering all possible variations of each valid ordering of N objects with up to and including i of them missing. For example, the partial order in Fig. 3(b) permits two linear extensions: (1 2 3 4) and (1 3 2 4). If the loss of any single object is tolerated, then the number of delivery orders that the destination could accept increases to eight: (1 2 3 4), (1 3 2 4) and (1 2 3), (1 2 4), (1 3 2), (1 3 4), (2 3 4), (3 2 4).

More precisely, if partial-order X has a single object, then $e_0(X) = e_1(X) = 1$ and $\forall_{i \geq 2}[e_i(X) = 0]$. (One cannot lose two or more objects from a partial order that only has one object.) If partial-orders X_1 and X_2 are combined in series and i objects can be lost, the resulting number of linear extensions is:

$$e_i(X_1; X_2) = \sum_{j=0}^{i} e_j(X_1) e_{i-j}(X_2). \tag{1}$$

Equation (1) sums all combinations of: losing i and 0 objects from partial-orders X_1 and X_2, respectively; plus losing $i-1$ and 1 objects from X_1 and X_2, respectively; ..., plus losing 0 and i objects from partial-orders X_1 and X_2, respectively. Analogously

$$e_i(X_1 \parallel X_2) = \sum_{j=0}^{i} \frac{(n_1 + n_2 - i)!}{(n_1 - j)!(n_2 - (i - j))!} e_j(X_1) e_{i-j}(X_2). \tag{2}$$

where by definition $i! = 1$ for $i \leq 1$.

The number of terms in a general formula for $e_i(X)$, where X is a composition (either series or parallel) of $k > 2$ partial orders, can be based on the number of *compositions* of i. *Compositions*[3] of i are expressions of i as a sum of positive integers with regard to order [6][4]. For instance, there are 8 *compositions* of the integer 4: $(1+1+1+1), (2+1+1), (1+2+1), (1+1+2), (2+2), (3+1), (1+3)$ and (4). Reference [2] presents a general formula of $e_i(X)$ based on the *partitions* of i; this formula is simplified here by considering *compositions*. The overall number of terms in the formula increases, but each

[3] Unfortunately, the term composition has two meanings. To minimize confusion, "composition" is used to refer to the combining either in series or in parallel of two or more partial orders. "*Composition*" (in italics) will refer to the mathematical concept of a set of integers summing to an integer i.

[4] Partitions of i do not take order into account.

one is simpler to represent. Thus by using *compositions*, the general formula is more readable.

Intuitively, when composing k partial orders, either in series or in parallel, with i losses, it is possible to have 1 loss in each of i partial orders and none in the others; or 2 losses in a single partial order, 1 loss in each of $i-2$ partial orders and none in the others; or 2 losses in each of 2 partial orders, 1 loss in each of $i-4$ partial orders and none in the others; ...; or i losses in 1 partial order and none in the others. If the ranges of summation variables are properly defined, all of the linear extensions that result from all of these *composition* possibilities are mutually exclusive. Thus in a general formula, there will be one term in the calculation of $e_i(X)$ for each *composition* of i. Each term itself must consider all possible combinations of partial orders in which the losses occur, thus resulting in multiple summations over all partial orders.

When partial orders are combined in series, one computes the product of the number of linear extensions of each partial order [with or without its permitted loss(es)] to compute the total number of possible ways the extensions can be combined. If the partial orders are combined in parallel, then one also must consider all possible interleavings of a single linear extension from each one. This results in an additional multinomial coefficient hereafter denoted $\mathcal{P}_{i,j}$ (coefficient for the j^{th} *composition* of i losses).

If $X_1,\ldots,X_k$ are k partial orders with $n_1,\ldots,n_k$ objects, respectively, then $X_1;\cdots;X_k$ and $X_1 \parallel \cdots \parallel X_k$ have $N=\sum_{j=1}^{k} n_j$ objects, and the formulae for e_1, e_2, and e_3 are as follows. A single formula is presented for composition either in series or in parallel. For series compositions, all $\mathcal{P}_{i,j}$ coefficients = 1; for parallel compositions, each $\mathcal{P}_{i,j}$ coefficient is given.

Several specific formulae are given in (3), (4), and (5) prior to the general formula for e_i in (6) to provide an intuition into the structure and complexity of the general formula.

$$e_1 = \sum_{i=1}^{k}\left(\mathcal{P}_{1,1}\; e_1(X_i)\prod_{\substack{j=1\\ j\neq i}}^{k} e_0(X_j)\right)$$

where
$$\mathcal{P}_{1,1} = \frac{(N-1)!}{(n_i-1)!\prod_{\substack{j=1\\ j\neq i}}^{k}(n_j!)} \tag{3}$$

$$e_2 = \sum_{i=1}^{k}\left(\mathcal{P}_{2,1}\; e_2(X_i)\prod_{\substack{j=1\\ j\neq i}}^{k} e_0(X_j)\right)$$

where
$$\mathcal{P}_{2,1} = \frac{(N-2)!}{(n_i-2)!\prod_{\substack{j=1\\ j\neq i}}^{k}(n_j!)} \tag{4}$$

$$+\sum_{i_1=1}^{k-1}\sum_{i_2=i_1+1}^{k}\left(\mathcal{P}_{2,2}\; e_1(X_{i_1})e_1(X_{i_2}) \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k} e_0(X_j)\right) \quad \text{if } (k\geq 2)$$

where
$$\mathcal{P}_{2,2} = \frac{(N-2)!}{(n_{i_1}-1)!(n_{i_2}-1)! \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k}(n_j!)}$$

The two terms for e_2 in (4) respectively represent all combinations of: 2 losses from 1 partial order and none from the others; and 1 loss from each of 2 partial orders and none from the others. In the second term, subscript i_2 takes on only those values greater than subscript i_1. This is to avoid counting twice the case of a single loss in each of X_{i_1} and X_{i_2}.

$$e_3 = \sum_{i=1}^{k}\left(\mathcal{P}_{3,1}\; e_3(X_i)\prod_{\substack{j=1\\ j\neq i}}^{k} e_0(X_j)\right)$$

where
$$\mathcal{P}_{3,1} = \frac{(N-3)!}{(n_i-3)!\prod_{\substack{j=1\\ j\neq i}}^{k}(n_j!)} \tag{5}$$

$$+\sum_{i_1=1}^{k-1}\sum_{i_2=i_1+1}^{k}\left(\mathcal{P}_{3,2}\; e_2(X_{i_1})e_1(X_{i_2}) \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k} e_0(X_j)\right) \quad \text{if } (k\geq 2)$$

where
$$\mathcal{P}_{3,2} = \frac{(N-3)!}{(n_{i_1}-2)!(n_{i_2}-1)! \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k}(n_j!)}$$

$$+\sum_{i_1=1}^{k-1}\sum_{i_2=i_1+1}^{k}\left(\mathcal{P}_{3,3}\; e_1(X_{i_1})e_2(X_{i_2}) \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k} e_0(X_j)\right) \quad \text{if } (k\geq 2)$$

where
$$\mathcal{P}_{3,3} = \frac{(N-3)!}{(n_{i_1}-1)!(n_{i_2}-2)! \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)}}^{k}(n_j!)}$$

$$+\sum_{i_1=1}^{k-2}\sum_{i_2=i_1+1}^{k-1}\sum_{i_3=i_2+1}^{k}\left(\mathcal{P}_{3,4}\; e_1(X_{i_1})e_1(X_{i_2})e_1(X_{i_3}) \cdot \prod_{\substack{j=1\\ (j\neq i_1)\wedge(j\neq i_2)\wedge(j\neq i_3)}}^{k} e_0(X_j)\right) \quad \text{if } (k\geq 3)$$

where

$$\mathcal{P}_{3,4} = \frac{(N-3)!}{(n_{i_1}-1)!(n_{i_2}-1)!(n_{i_3}-1)! \prod_{\substack{j=1 \\ (j\neq i_1)\wedge(j\neq i_2)\wedge(j\neq i_3)}}^{k} (n_j!)}$$

These formulae can be generalized by a single formula. Let

$$e_i(X_1;\ldots;X_k) = \sum_{t=1}^{\#Comp(i)} Term_{i,t} \quad (\text{with } \mathcal{P}_{i,t}=1)$$

$$e_i(X_1 \parallel \ldots \parallel X_k) = \sum_{t=1}^{\#Comp(i)} Term_{i,t} \quad (\text{with } \mathcal{P}_{i,t} \text{ given})$$

where $\#Comp(i)$ = the number of *compositions* of i, and $Term_{i,t}$ is defined below. Each term for a series composition assumes the coefficient $\mathcal{P}_{i,t} = 1$. For parallel compositions, the $\mathcal{P}_{i,t}$ values are given.

$Term_{i,t}$ is based on the tth *composition* of i and is calculated as follows. Let the tth *composition* consist of b integers denoted $\lambda = \{\lambda_1, \ldots, \lambda_b\}$. That is, $\sum_{j=1}^{b} \lambda_j = i$. For example, one *composition* of the integer 21 is $\lambda = (8+4+4+2+2+1)$.

Each $Term_{i,t}$ has b summations where b partial orders "contribute" the lost objects and $k-b$ partial orders contribute no loss.

$$Term_{i,t} = \sum_{i_1=1}^{k-b+1} \sum_{i_2=i_1+1}^{k-b+2} \sum_{i_3=i_2+1}^{k-b+3} \cdots \sum_{i_{b-1}=i_{b-2}+1}^{k-1} \sum_{i_b=i_{b-1}+1}^{k} \cdot \left(\mathcal{P}_{i,t} \prod_{u=1}^{b} e_{\lambda_u}(X_{i_u}) \prod_{\substack{j=1 \\ (j\neq i_1)\wedge(j\neq i_2)\wedge\cdots\wedge(j\neq i_b)}}^{k} e_0(X_j) \right) \quad \text{if}(k \geq b) \qquad (6)$$

where

$$\mathcal{P}_{i,t} = \frac{(N-i)!}{\prod_{u=1}^{b} (n_{i_u} - \lambda_u)! \prod_{\substack{j=1 \\ (j\neq i_1)\wedge(j\neq i_2)\wedge\cdots\wedge(j\neq i_b)}}^{k} (n_j!)}.$$

For an ordered or unordered service with N objects, e_i reduces to $\mathbf{C}_N^{N-i}(=\frac{N!}{i!(N-i)!})$ and $\mathbf{P}_N^{N-i}(=\frac{N!}{i!})$, respectively, where $\mathbf{C}$ and $\mathbf{P}$ represent combinations and permutations.

From a computational point of view, computing e_i for a partial order composed of multiple smaller partial orders is simplified by composing them two at a time and repeatedly using the formulae (1) and (2). The computational complexity of computing e_i is discussed further in [11].

TABLE I
STATISTICS FOR PARTIAL ORDER IN FIG. 2

i	*Ordered*$(\mathbf{C}_N^{N-i})$	e_i	*Unordered*$(\mathbf{P}_N^{N-i})$	M_i	m_i
0	1	5,417,717,760	121,645,100,408,832,000	.5697	.5697
1	19	23,381,729,280	121,645,100,408,832,000	.6016	.6069
2	171	49,662,412,800	60,822,550,204,416,000	.6233	.6373
3	969	67,881,748,480	20,274,183,401,472,000	.6376	.6642
4	3,876	66,369,896,320	5,068,545,850,368,000	.6466	.6891
5	11,628	49,056,157,696	1,013,709,170,073,600	.6517	.7124
6	27,132	28,344,600,128	168,951,528,345,600	.6543	.7347
7	50,388	13,092,017,600	24,135,932,620,800	.6554	.7560
8	75,582	4,912,613,408	3,016,991,577,600	.6558	.7766
9	92,378	1,516,456,832	335,221,286,400	.6559	.7966
10	92,378	389,093,456	33,522,128,640	.6559	.8161
11	75,582	83,725,808	3,047,466,240	.6559	.8354
12	50,388	15,226,920	253,955,520	.6559	.8546
13	27,132	2,355,648	19,535,040	.6559	.8740
14	11,628	311,452	1,395,360	.6559	.8940
15	3,876	35,268	93,024	.6559	.9152
16	969	3,414	5,814	.6559	.9386
17	171	280	342	.6559	.9657
18	19	19	19	.6559	1.0000
19	1	1	1	—	—

C. Comparing Partial-Order Services

Using arbitrary precision arithmetic routines, programs were developed to compute e_i values for an arbitrary series-parallel partial order. Table I indicates e_i values for $0 \leq i < N$ for the Anatomy and Physiology Instructor example in Fig. 2.

Additionally, the corresponding number of linear extensions for an ordered and unordered service are tabulated. For example, if no losses are permitted, there are 5,417,717,760 valid orderings (i.e., linear extensions) out of the total possible 19! (=121,645,100,408,832,000) orderings, a fraction of $4.453 * 10^{-8}$.

On comparing over five billion valid orderings in a partial-order service with just one valid ordering in an ordered service, the partial order seems quite flexible, yet a fraction on the order of 10^{-8} hardly seems to reflect this. The significance is visualized more easily by considering the number of valid orderings on a normalized logarithmic scale. Therefore the following normalized partial-order metrics in the interval [0,1] are proposed where 0 represents reliable ordered service, values from 0 to 1 represent increasingly more flexible partial reliable, partial-order services, and 1 represents unreliable unordered service. For partial-order X containing N objects and considering a service with i losses:

$$m_i(X) = \frac{\log(e_i(X))}{\log(\mathbf{P}_N^{N-i})} \qquad for \quad 0 \leq i < N$$

$$M_i(X) = \frac{\log \sum_{j=0}^{i}(e_j(X))}{\log \sum_{j=0}^{i}(\mathbf{P}_N^{N-j})} \qquad for \quad 0 \leq i < N$$

The metric $m_i(X)$ represents a relative comparison between the number of permitted extensions of a partial order and an unordered service for N objects and **exactly** i losses. The metric $M_i(X)$ represents a similar relative comparison, but for i **or fewer** losses. The m_i and M_i values for the multimedia example in Fig. 2 also are tabulated in Table I.

These metrics provide better insight into this partial order's flexibility than do the e_i values. The metrics m_i and M_i allow one to quantify and compare partial orders with respect to communication constraints independent of N. Consider,

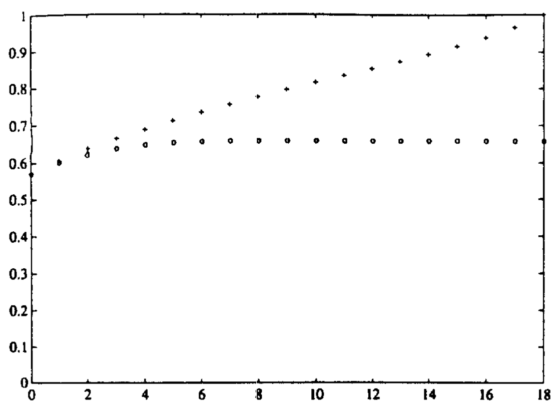


Fig. 8. m_i (+) and M_i (o) versus i values for partial order in Fig. 2.

for example, Fig. 8 which plots the full set of m_i and M_i values for Fig. 2. As more losses are permitted, the m_i values converge to those of unordered service (for which all m_i values equal 1.) For a partial order, the M_i values increase, but never reach 1 because the partial order constraints always place limitations of the set of valid orderings.

This example leads to the conjecture that as the number of tolerated losses increases, U-PO service never decreases in flexibility relative to an unordered service that tolerates the same number of losses. That is, both $m_i(X)$ and $M_i(X)$ are nondecreasing functions of i;

$$\forall_{0 \leq i < (N-1)} [m_i(X) \leq m_{i+1}(X)]$$

$$\forall_{0 \leq i < (N-1)} [M_i(X) \leq M_{i+1}(X)]$$

If this conjecture were not true, then in some cases, one would be increasingly more motivated to use an ordered service for an application having only partial-order constraints as toleration for loss increased. This seems counterintuitive.

Similarly, study of m_i and M_i demonstrates that for $i \neq j, m_i(X_1) < m_i(X_2)$ does not imply $m_j(X_1) \leq m_j(X_2)$ (and analogously for M_i). Merely because partial-order A is more flexible than partial-order B when i (or fewer) losses can be tolerated, A may be less flexible than B when $j > i$ (or fewer) losses can be tolerated (contrary to the authors' initial intuition).

The above formulae for e_i assume that all objects are equivalent from the viewpoint of loss. In some applications, however, this may not be true (see Section IV–B). For example, in Fig. 2, perhaps one could tolerate the loss of any single pair of associated parallel objects in 1 through 12 [i.e., (1,2) or (3,4) or ... or (11,12)], but not any single or any random two of the nineteen objects. The previous formula for calculating e_2 allows any two objects to be lost and thus overestimates the number of valid linear extensions with this constraint.

One can take into account such restrictions when computing any e_i value. In this example, one can recompute the e_2 value for just the partial order consisting of objects 1-12, and define $e_2 = 0$ for the other five parallel composed partial orders (objects 13-19). Additionally, this constraint implies $e_1 = 0$ for all six parallelly composed partial orders. With this particular constraint, the number of linear extensions is 6,273,146,880; less than 15% of the general e_2 value of 49,662,412,800.

IV. Estelle Protocol Specification

While calculating $e_i(P)$ is useful for evaluating and comparing partial orders, it remains a practical problem for a destination to determine as objects arrive if they are in one of the valid orders as defined by P. That is, is the arriving order a member of $L(P)$? If not, arriving objects must be buffered to guarantee the particular partial order in agreement at the time.

Enumerating $L(P)$ is equivalent to finding all possible topological sortings for a given partial order [18]. Fortunately in practice, a destination need not enumerate $L(P)$ to decide if an arriving object can be delivered. The destination merely needs to see if the arriving object satisfies the defined partial order.

Once a source/destination pair have agreed on the partial order in question, there remain two problems: 1) what protocol does the destination use to evaluate an arriving object's validity with regards to order, and 2) when/how does a destination decide that an object which has not yet arrived can be presumed lost?

A. Protocol POC

Our protocol, entitled Partial_Order_Connection (POC) dynamically updates its information each time an object arrives. It is specified in the language Estelle (see Appendix A), an ISO International Standard Formal Description Technique for specifying communication services/protocols and, more generally, distributed systems [8], [17]. The specification has been designed and validated using several formal description tools: Pet-Dingo, a portable Estelle translator and distributed generator for simulations [27]; and GROPE, a simulation system that provides graphical animation to visualize an Estelle specification [5].

Since it makes no practical sense to put a POC on top of a service that is already fully ordered and fully reliable, Protocol POC expects that the underlying network service is unreliable. It will lose and duplicate objects, and sometimes deliver them out of the order transmitted. In all cases (R-PO and U-PO), Protocol POC will remove duplicates.

The sender transmits (possibly repeating periods of) N objects using at most NUM_SND_BUFFERS to remember unacknowledged objects outstanding at any moment in time. The receiver is assumed to have NUM_RCV_BUFFERS with which to temporarily store out of order objects. In case of repeating periods, the sender and receiver distinguish identically numbered objects from different periods by a period number.

B. Object Reliability Classes

In Section III's discussion of U-PO service, all objects are equal with regards to their reliability. This classification is reasonable if all objects are identical (e.g., video frames in a 30 frame/s film). Applications that require a partial-order service, however, may contain a variety of object types. Thus, Protocol POC defines three object reliability classes within a U-PO service: BART-NL, BART-L, NBART-L, where it is the application's responsibility to define which object belongs to which class[5]. While classic transport services generally treat all objects equally, the sending and receiving functions of Protocol POC behave differently for each class of object.

BART-NL objects must be delivered to the destination. These objects have long temporal value that lasts for an entire established connection and require reliable delivery. If all objects are of type BART-NL, the service is R-PO service. An example of BART-NL objects would be the windows in the screen refresh Example 2 of Section II. To assure eventual delivery of a BART-NL object in Protocol POC, the sender buffers it, starts a timeout timer, and retransmits it if no ack arrives before the timeout. The receiver in turn returns an ack when the object has safely arrived and been delivered or buffered.

BART-L objects have temporal value over some intermediate amount of time, enough to permit timeout and retransmission, but not everlasting. Once the temporal value of these objects has expired, it is better to presume them lost than to delay further the delivery pipeline of information. One possibility for deciding when an object's usefulness has expired is to require each object to contain information defining its precise temporal value [14]. An example of a BART-L object would be a movie subtitle which is to be displayed during a twenty second film sequence. If not delivered sometime during the first ten seconds, the subtitle loses its value and can be presumed lost. In Protocol POC, these objects are buffered-acked-retransmitted up to a certain point in time and then presumed lost.

NBART-L objects are those associated with strict real-time applications. Their temporal values are too short to bother timing out and retransmitting. An example of a NBART-L object might be a single packet of speech in a packetized phone conversation or one image in a 30 image/s film. In Protocol POC, a sender transmits these objects once, and the service makes a best effort to deliver them. If the one attempt is unsuccessful, no further attempts are made.

Protocol POC's general architecture is shown in Appendix A. A User_Sender (e.g., sending application) supplies objects to the POC_Sender according to the partial order, not necessarily in sequence order $1, 2, \ldots, N, 1, 2, \ldots$ The partial order defines both the possible orders of transmission by the sending application and the orders of delivery to the receiving application. The POC_Sender buffers and, if necessary, retransmits any BART-NL or BART-L objects that are not acknowledged within a predefined timeout period. The total number of unacknowledged BART-NL and BART-L objects never exceeds {NUM_SND_BUFFERS}.

Each time an object arrives at the receiver, Estelle transition Check_Newly_Arriving_Object becomes firable. If the object is within the receiver's window and is not a duplicate, it is either immediately delivered to the User_Receiver (e.g, destination application) or, if not deliverable according to the partial order, stored for future delivery. BART-NL and BART-L objects are then acked. Out-of-partial-order objects for which there is no available buffer space simply are discarded. Whenever an object is delivered to the User_Receiver, transition Check_Buffers_For_Delivery becomes enabled and checks all occupied receive buffers to see if the just delivered object now enables the delivery of any stored objects.

Due to practical page constraints, the Estelle specification in Appendix A is abbreviated only to include the architecture and data transfer phase. It is assumed that a connection already has been established, and that an initial partial order and vector defining the reliability class of each of the N objects has been negotiated.

The full data transfer phase allows the POC_Sender and POC_Receiver to change the partial order dynamically. Dynamic changes are to be permitted although the POC_Sender and POC_Receiver are obligated to complete one partial order before beginning another. A sender and receiver may not handle multiple different partial orders simultaneously. Currently the authors predict any gain in performance would be minor and not worth the added complexity needed to permit multiple orders.

Any partial order can be represented in $N(N-1)/2$ bits as an NXN upper-triangular matrix where N is the number of objects in the partial order [1]. If the partial order is series-parallel, it can be represented as the intersection of two total orders [29]. By assuming one total order to be $1, 2, \ldots, N$, a series-parallel partial order can be encoded in $N \log N$ bits.

For a U-PO service with BART-L and NBART-L objects, a POC_Receiver can decide at any time that an object is *presumed lost* and then continue delivering objects as if the lost one had been delivered. This represents the situation where a multimedia application decides that an object has lost its temporal value. To decide when to presume an object is lost, POC_Receiver includes transition Validate_Temporal_Value to regularly check if delivery to the User_Receiver of each expected object in the reception window is still worthwhile.

As soon as an expected BART-L or NBART-L object's temporal value expires as determined by a call to a special function Is_Object_Still_Useful, the object is presumed lost. Then all currently buffered objects are checked to see if their delivery is now enabled. Should an object that was presumed lost arrive later, it will be discarded since it is no longer of any value, and if type BART-L, it will be acknowledged to stop its retransmission by the sender. Thus for this protocol, an ack is sent any time a BART-L object is delivered, stored, or presumed lost; as a result, it is possible to ack an object that has not yet been sent.

The details of Is_Object_Still_Useful are not defined in Appendix A. This function can be internal to Protocol POC, in which case each object is required to contain information defining its precise temporal value. Otherwise this function must contact the User_Receiver to decide when an object is no longer valuable. The latter approach requires coordination between the User_Receiver and the POC_Receiver.

In regards to the metrics discussed in Section III, when the set of presumed losses exceeds a defined limit as determined by a function assumed to have been negotiated at connection establishment, a message is sent to the User_Receiver indicating the negotiated QOS is not being provided. It is then up

[5]BART stands for (Buffers, Acks, Retransmissions, Timeouts), four mechanisms employed to obtain reliability. L indicates that loss is permitted; NL indicates no loss is allowed.

to this user to determine whether or not to continue with the partial-order service.

It is emphasized that while closed-form formulae for computing $m_i(X)$ and $M_i(X)$ exactly are provided only for series-parallel partial orders, Protocol POC is applicable for *any* partial order, not only those that are series-parallel. When a partial order contains BART-NL objects, the e_i values are reduced since certain linear extensions with loss are no longer permitted. Calculation of e_i in this case uses the same formulae derived in Section III with the single difference that initial values $e_1(X)$ equal 0, not 1, whenever X is a partial order representing a single BART-NL object.

V. Conclusion

This work 1) introduces and motivates a partial order, partial reliable transport service/protocol, 2) investigates the definition and calculation of metrics for quantifying a partial-order service, and 3) provides a formal Estelle specification of a protocol that provides partial-order service. The authors currently are simulating Protocol POC using OPNET, a networking simulation system, and implementing a partial-order version of TCP based on a submitted RFC [10]. The simulation and empirical studies will evaluate more precisely the expected delay/memory/bandwidth performance improvements compared to an ordered service for various combinations of 1) different partial orders and loss tolerances (i.e., different m_i and M_i values), 2) different distributions of disorder and loss supplied by the underlying service, and 3) different sender–receiver window sizes. The goal is to better understand the potential performance gains when using a partial-order service over the full range of unreliable network services.

Appendix

Estelle Specification of Protocol: Partial-Order Connection

```
specification PartialOrderConnection systemactivity; default individual queue; timescale millisecond;

{        +----------------+      +----------------+
         | User_Sender    |      | User_Receiver  |
         +-------+--------+      +-------+--------+
                 |ucep              ucep|
                 |ucep              ucep|
         +-------+--------+      +-------+--------+        General Architecture
         |  POC_Sender    |      |  POC_Receiver  |
         +-------+--------+      +-------+--------+
                 |lcep              lcep|
               · |lcep1             lcep2|
         +-------+-----------------------+-------+
         |                Network                |
         +---------------------------------------+                                                  }

const N                        = ...;          { # tsdus in negotiated PO per period                 }
      NUM_PARTIAL_ORDER_BITS   = (N*(N-1)/2)   { # bits to encode partial order                      }
      NUM_DATA_PER_N_OR_T_SDU  = ...;          { info data bytes per nsdu or tsdu                    }
      NUM_RCV_BUFFERS          = ...;          { # of receive buffers                                }
      MAX_PERIOD_PER_RW        = ...;          { = ceiling((NUM_RCV_BUFFERS - 1)/N) + 1              }
      MAX_PERIOD_PER_RW_MINUS1 = (MAX_PERIOD_PER_RW - 1)
      NUM_SND_BUFFERS          = ...;          { # of send buffers                                   }
      SIZE_OF_NEEDS_ACK        = ...;          { >= 2* max # periods before needs_ack array cycles   }
      SIZE_OF_NEEDS_ACK_MINUS1 = (SIZE_OF NEEDS_ACK - 1)
      EMPTY                    = -1;           { represents empty/null snd/rcv buffer                }
      ACK_TIMEOUT              = ...;          { sender's timeout before retransmissions             }
      CHECK_VALIDITY_INTERVAL  = ...;          { timeout for check if objects in rcv buffers have temp value }
type sdu_info_type             = array [1..NUM_DATA_PER_N_OR_T_SDU] of integer;
     tsdu_type                 = record seqnum, period: integer; info: sdu_info_type; end;
     nsdu_type                 = record header: integer; seqnum, period: integer; info: sdu_info_type; end;
     ident_type                = record period, seqnum: integer; end;
     PO_matrix_type            = array [1..N,1..N] of 0..1;
     partial_order_type        = array [1..NUM_PARTIAL_ORDER_BITS] of integer;
     reliability_type          = (NBART_L,BART_NL,BART_L);
     partial_reliability_type  = array [1..N] of reliability_type;
     array_1_N_of_boolean_type = array [1..N] of boolean;
channel tsdu_channel(usr,pvd); { connects user/application and partial_order_service                 }
   by usr: t_data_req(tsdu: tsdu_type);
   by pvd: t_data_ind(tsdu: tsdu_type); QOSLoss_failed; po_not_respected(period,seqnum: integer);
channel nsdu_channel(usr,pvd); { connects partial_order_service and network                          }
   by usr,pvd: n_data_req(nsdu: nsdu_type); n_data_ind(nsdu: nsdu_type);
               ack(period: integer; seqnum: integer);
```

```
{ ---------------------- Application: Transport Service User ------------------------------------------- }
module User_type activity;  ip ucep: tsdu_channel (usr);  end;
body User_Sender_body for User_type; external;  { sends tsdus to the transport protocol                  }
body User_Receiver_body for User_type; external; { receives tsdus from the transport protocol            }

{ --------------------- PO Transport Protocol --------------------------------------------------------- }
module POC_type activity; ip ucep: tsdu_channel(pvd); lcep: nsdu_channel(usr); end;
body POC_Sender_Body for POC_Type;
state active;
type  needs_ack_type     = array [1..N,0..SIZE_OF_NEEDS_ACK_MINUS1] of boolean;
      snd_buffers_type   = array [1 .. NUM_SND_BUFFERS] of tsdu_type;
var nsdu               : nsdu_type;
    snd_PO_matrix      : PO_matrix_type;
    partial_order      : partial_order_type;
    PR_vector          : partial_reliability_type;  { vector describing partial reliability          }
    partial_reliability: partial_reliability_type;  { negotiated partial reliability                 }
    snd_used_buffers   : integer;                   { # send buffers currently in use                }
    snd_buffers        : snd_buffers_type;          { contains sent BART tsdus awaiting ack          }
    needs_ack          : needs_ack_type;   { true if tsdu i of per. [j mod SIZE_OF_NEEDS_ACK] awaits ack }
    earliest_per       : integer;          { period represented by first column of needs_ack         }
    snd_curr_per       : integer;          { period for which tsdu are being sent;                   }
                                           { not necessarily 1st period of objects awaiting ack      }
    num_curr_per_tsdu_sent: integer;       { num tsdus of current period already sent once           }

procedure FreeSndBuffer(period, seqnum: integer; var snd_buffers: snd_buffers_type;
                        var snd_used_buffers: integer);  primitive;
          { find stored tsdu(period,tsdu) and free the buffer                                       }
procedure Init_PO_Matrix(var snd_PO_matrix: PO_matrix_type; partial_order: partial_order_type); primitive;
          { initialize upper right triangular snd_PO_matrix with negotiated partial order          }
procedure Init_PR_Vector(var PR_vector      : partial_reliability_type;
                         partial_reliability: partial_reliability_type); primitive;
          { initialize PR_vector with negotiated partial_reliability                               }

function IsObjectInSndBuffers(period: integer; seqnum: integer): boolean; primitive;
          { true when tsdu (period,seqnum) is in one of the snd buffers                             }
function IsPORespected (period: integer; seqnum: integer): boolean; primitive;
          { true when given tsdu respects partial order.                                            }
function IsObjectSendable(period: integer; seqnum: integer): boolean; primitive;
          { returns true when tsdu(period,seqnum) can be sent for the first time                    }
procedure Store_Sent_tsdu(tsdu: tsdu_type; var snd_buffers: snd_buffers_type;
                          var snd_used_buffers: integer); primitive;
          { find an empty buffer in snd_buffers and store tsdu in it                                }
procedure Update_Needs_Ack(var needs_ack: needs_ack_type; var earliest_per: integer;
                           period, seqnum: integer); primitive;
          { when ack arrives, update knowledge of outstanding acks                                  }
procedure Update_Snd_PO_Matrix(var snd_PO_matrix: PO_matrix_type; var num_curr_per_tsdu_sent: integer;
                               var snd_curr_per: integer; seqnum: integer); primitive;
          { each time an object is sent, the dynamic PO matrix has to be updated                    }
procedure Prepare_nsdu(var nsdu: nsdu_type; tsdu: tsdu_type); primitive;
          { use info in tsdu to form nsdu                                                           }

initialize to active
    var i,j: integer;
    begin
    snd_curr_per := 1;
    num_curr_per_tsdu_sent := 0;
    snd_used_buffers := 0;
    for i := 1 to N do for j := 0 to SIZE_OF_NEEDS_ACK_MINUS1 do needs_ack[i,j] := true;
    earliest_per := 1;
    for i := 1 to NUM_SND_BUFFERS do
        begin
        snd_buffers[i].period := EMPTY;
        snd_buffers[i].seqnum := EMPTY;
        for j := 1 to NUM_DATA_PER_N_OR_T_SDU do snd_buffers[i].info[j] := EMPTY;
        end;
    Init_PO_Matrix(snd_PO_matrix,partial_order);
    Init_PR_Vector(PR_vector,partial_reliability);
    end;

trans
from active to active         { User_Sender wishes to transmit an object to User_Receiver                   }
when ucep.t_data_req(tsdu)
provided IsObjectSendable(tsdu.period,tsdu.seqnum)
    name Data_Request:
    begin
    Update_Snd_PO_Matrix(snd_PO_matrix, num_curr_per_tsdu_sent,snd_curr_per,tsdu.seqnum);
    if (needs_ack[tsdu.seqnum, tsdu.period mod SIZE_OF_NEEDS_ACK]) then
        begin
        Prepare_nsdu(nsdu,tsdu);        { nsdu is based upon info within given tsdu                  }
        output lcep.n_data_req(nsdu); { transmit the object                                          }
        { buffer all BART objects in case later retransmission is needed                             }
        if (PR_vector[tsdu.seqnum] = BART_L) or (PR_vector[tsdu.seqnum] = BART_NL) then
            Store_Sent_tsdu(tsdu,snd_buffers,snd_used_buffers)
        else  { assert: PR = NBART_L;  by def, on sending NBART_L object, no need to wait for ack    }
            Update_Needs_Ack(needs_ack,earliest_per,tsdu.period,tsdu.seqnum);
        end;
    { else tsdu has been previously declared lost; it is not sent                                    }
    end;

from active to active       { User_Sender tries to transmit an object not according to the partial order  }
when ucep.t_data_req(tsdu)
provided not(IsPORespected(tsdu.period,tsdu.seqnum))
    name Error_in_Data_Request:
    begin { user is warned it did not respect the negotiated partial order                           }
    output ucep.po_not_respected(tsdu.period,tsdu.seqnum)
    end;

from active to active        { An ack returned from the User_Receiver arrives from the Network             }
when lcep.ack(period,seqnum)
    name Ack_Management:
    begin
    if IsObjectInSndBuffers(period,seqnum) then
        begin
        Update_Needs_Ack(needs_ack,earliest_per,period,seqnum);
        FreeSndBuffer(period,seqnum,snd_buffers,snd_used_buffers);
        end
    else
        begin { ack is either duplicate ack or ack of a declared lost tsdu not yet sent              }
        { check if ack is in one of the periods being monitored by needs_ack array                   }
        if (earliest_per <= period) and (period < (earliest_per + SIZE_OF_NEEDS_ACK)) then
           Update_Needs_Ack(needs_ack,earliest_per,period,seqnum);
        { when (snd_curr_per < period) then ack = duplicate; discard it                              }
        end;
    end;

from active to active         { Retransmit objects if expected ack has not arrived                          }
any X: 1..NUM_SND_BUFFERS do
provided snd_buffers[X].period <> EMPTY
delay (ACK_TIMEOUT)
    name Timeout_Retransmit:
    begin
    Prepare_nsdu(nsdu,snd_buffers[X]);
    output lcep.n_data_req(nsdu);
    end;
end;

{ ------------------------------------------------------------------------------------------------------ }
body POC_Receiver_body for POC_type;
state   active;
type rcv_buffers_type = array [1..NUM_RCV_BUFFERS] of tsdu_type;
     stored_type      = array [0..MAX_PERIOD_PER_RW_MINUS1,1..N] of integer;
     rcv_proc_obj_event_type   = (LOSE_OBJ, DELIVER_NEW_OBJ, DELIVER_BUF_OBJ);
     rcv_adj_per_event_type    = (OBJ_PROCESSED, OBJ_STORED, INIT_EDGES);
var tsdu: tsdu_type;
    rcv_used_buffers   : integer;                 { # rcv buffers currently filled                        }
    rcv_PO_matrix      : PO_matrix_type;          { dynamic partial order matrix                          }
    partial_order      : partial_order_type;      { negotiated static partial order                       }
    PR_vector          : partial_reliability_type; { dynamic vector describing partial reliability        }
    partial_reliability: partial_reliability_type; { negotiated partial reliability                       }
    delivered     : array_1_N_of_boolean_type;    { true if tsdu[i] was delivered                         }
    lost          : array_1_N_of_boolean_type;    { true if tsdu[i] assumed lost                          }
    rcv_buffers   : rcv_buffers_type;             { buffers for out of order tsdus                        }
    check_buffers : boolean;         { if true, check stored tsdus for possible delivery                  }
    rcv_curr_per  : integer;         { period of objs being delivered                                     }
    stored        : stored_type;     { indicates buffer location of stored tsdu                           }
    num_buffable_per  : integer;     { # successive periods which may be fully or partially buffered      }
    last_buffable_per : integer;     { last period for which objects may be buffered                      }
    rcv_proc_obj_event: rcv_proc_obj_event_type;   { receiver events                                      }
    rcv_adj_per_event : rcv_adj_per_event_type;
    num_obj_curr_per_deliv_lost   : integer;  { # objects in current period already deliv'd or lost       }
    max_num_obj_last_buffable_per : integer;  { max objects in last_bufferable_per                        }
    curr_num_obj_last_buffable_per: integer;  { current objects in last_buffable_per                      }
```

```
function IsObjectStillUseful(seqnum: integer): boolean; primitive;
          { return true if particular object still has temporal value; otherwise false                 }
function IsQOSLossExceeded: boolean; primitive;
          { true if most recent loss results in less than negotiated QOS                               }
function IsObjectDeliverable (rcv_PO_matrix: PO_matrix_type; tsdu: tsdu_type): boolean; primitive;
          { true when tsdu respects partial order and can be immediately delivered                     }
procedure Init_PO_Matrix(var rcv_PO_matrix: PO_matrix_type; partial_order: partial_order_type); primitive;
          { initialize PO_matrix with negotiated partial order                                         }
procedure Init_PR_Vector(var PR_vector      : partial_reliability_type;
                         partial_reliability: partial_reliability_type); primitive;
          { initialize PR_vector with negotiated partial reliability class information                 }
procedure Store_Received_tsdu(tsdu: tsdu_type; var rcv_buffers: rcv_buffers_type;
                              var rcv_used_buffers: integer); primitive;
          { find an empty buffer in rcv_buffers and store tsdu in it                                   }
procedure Process_Object(seqnum: integer; rcv_proc_obj_event: rcv_proc_obj_event_type); primitive;
          { object has been: delivered immediately upon arrival (DELIVER_NEW_OBJ) or delivered from    }
          { a buffer (DELIVER_BUF_OBJ) or presumed lost (LOSE_OBJ); perform needed processing          }
function NumUndelivableObjects(matrix: PO_matrix_type): integer; primitive;
          { determine # of buffers to reserve for objects in current period                            }
function InBuffer(tsdu: tsdu_type): boolean; primitive;
          { Determine if object is currently being buffered                                            }
function AlreadyProcessed(tsdu: tsdu_type): boolean; primitive;
          { check if object is A. Before receiving current period; B. In receiving current period and  }
          { either Delivered or lost; or C. Buffered                                                   }
function IsObjectBufferable(tsdu: tsdu_type): boolean; primitive;
          { check if object can be buffered                                                            }
function IsObjectReceivable(rcv_PO_matrix: PO_matrix_type; tsdu: tsdu_type): boolean;  primitive;
          { check if object is either deliverable or bufferable                                        }
procedure Adjust_Last_Period(rcv_adj_per_event: rcv_adj_per_event_type; period: integer); primitive;
          {keep track of latest period for which an object can be buffered                             }
procedure Prepare_tsdu( nsdu: nsdu_type; var tsdu: tsdu_type); primitive;
          { extract info from nsdu to produce tsdu; no need to extract nsdu header                     }

initialize to active
   var i,j: integer;
   begin
   num_buffable_per := 0;
   check_buffers := false;
   rcv_used_buffers := 0;
   for i:= 1 to NUM_RCV_BUFFERS do
       begin
       rcv_buffers[i].period := EMPTY;
       rcv_buffers[i].seqnum := EMPTY;
       for j:=1 to NUM_DATA_PER_N_OR_T_SDU do rcv_buffers[i].info[j] := EMPTY;
       end;
   for i := 1 to N do
       begin  .
       for j := 0 to MAX_PERIOD_PER_RW_MINUS1 do stored[j,i] := EMPTY;;
       lost[i] := false;
       delivered[i] := false;
       end;
   Init_PO_Matrix(rcv_PO_matrix,partial_order);
   Init_PR_Vector(PR_vector,partial_reliability);
   rcv_curr_per := 1;
   num_obj_curr_per_deliv_lost    := 0;
   curr_num_obj_last_buffable_per := 0;
   max_num_obj_last_buffable_per  := 0;
   Adjust_Last_Period(INIT_EDGES,rcv_curr_per);
   end;

trans
from active to active          { An object from User_Sender arrives from the network                        }
when lcep.n_data_ind(nsdu)
   name Check_Newly_Arriving_Object:
   begin
   Prepare_tsdu(nsdu,tsdu);     { extract important info from arriving nsdu                                 }
   if IsObjectReceivable(rcv_PO_matrix,tsdu) then
       begin
       if IsObjectDeliverable(rcv_PO_matrix,tsdu) then
          begin { deliver the tsdu                                                                         }
          output ucep.t_data_ind(tsdu);
          Process_Object(tsdu.seqnum,DELIVER_NEW_OBJ);
          Adjust_Last_Period(OBJ_PROCESSED,tsdu.period);
          end
```

```
        else
            begin
            Store_Received_tsdu(tsdu,rcv_buffers,rcv_used_buffers);
            Adjust_Last_Period(OBJ_STORED,tsdu.period);
            end;
        if (PR_vector[tsdu.seqnum] <> NBART_L) then
            output lcep.ack(tsdu.period,tsdu.seqnum);
        end
    else
        { cannot deliver nor buffer; send ack if a duplicate                                  }
        begin
        if AlreadyProcessed(tsdu) and (PR_vector[tsdu.seqnum] <> NBART_L) then
            output lcep.ack(nsdu.period,nsdu.seqnum);
        end;
    end;

from active to active    { Check if any currently buffered objects can be delivered to User_Receiver        }
provided check_buffers and (1 <= rcv_used_buffers)
    { enabled when an object delivered or lost provided at least one buffer is full                         }
    var i, buf_number: integer;
    name Check_Buffers_For_Delivery:
    begin
    while check_buffers do
        begin { loop until one complete loop fails to deliver an object                                  }
        check_buffers := false;
        for i := 1 to N do
            begin
            buf_number := stored[rcv_curr_per mod MAX_PERIOD_PER_RW,i];
            if (buf_number <> EMPTY) then
                if IsObjectDeliverable(rcv_PO_matrix,rcv_buffers[buf_number]) then
                    begin { a deliverable buffered object has been found        }
                    output ucep.t_data_ind(rcv_buffers[buf_number]);
                    Process_Object(rcv_buffers[buf_number].seqnum,DELIVER_BUF_OBJ);
                    Adjust_Last_Period(OBJ_PROCESSED,rcv_buffers[buf_number].period);
                    end;
            end;
        end;
    end;

from active to active           { Periodically check objects for temporal value                         }
delay (CHECK_VALIDITY_INTERVAL)
    var i: integer;
    name Validate_Temporal_Value:
    begin
    for i := 1 to N do
        if (not (delivered[i] or lost[i] or IsObjectStillUseful(i))) then
            begin { found nonuseful object; presume it lost                                            }
            Process_Object(i,LOSE_OBJ);
            if IsQOSLossExceeded then output ucep.QOSLoss_failed;
            if PR_vector[i] = BART_L then output lcep.ack(rcv_curr_per,i);
            Adjust_Last_Period(OBJ_PROCESSED,rcv_curr_per);
            end;
    end;
end;

{ --------------------- Network Layer ------------------------------------------------------------------- }
module Network_type activity; ip lcep1: nsdu_channel(pvd); lcep2: nsdu_channel(pvd); end;
body Network_body for Network_type; external; { network service between transport protocol entities      }

{ --------------------- Main Specification --------------------------------------------------------------- }
modvar  User_Sender,User_Receiver: User_type; POC_Sender, POC_Receiver: POC_type; Network: Network_type;
initialize
    begin
    init User_Sender   with User_Sender_body;
    init User_Receiver with User_Receiver_body;
    init POC_Sender    with POC_Sender_Body;
    init POC_Receiver  with POC_Receiver_body;
    init Network       with Network_body;
    connect User_Sender.ucep   to POC_Sender.ucep;
    connect User_Receiver.ucep to POC_Receiver.ucep;
    connect POC_Sender.lcep    to Network.lcep1;
    connect POC_Receiver.lcep  to Network.lcep2;
    end;
end.
```

ACKNOWLEDGMENT

The authors gratefully thank the anonymous referees for their substantive suggestions.

REFERENCES

[1] P. D. Amer, C. Chassot, T. Connolly, and M. Diaz, "Partial order transport service for multimedia applications: Reliable service." in *Proc 2nd High Perform. Distrib. Comput. Conf. (HPDC)*, Spokane, WA, July 1993, pp. 272–280.

[2] ——, "Partial order transport service for multimedia applications: Unreliable channels," in *Proc 3rd Int. Networking Conf. (INET)*, BFA 1-10, San Francisco, CA, Aug 1993.

[3] D. Anderson and G. Homsy, "A continuous media I/O server and its synchronization mechanism," *IEEE Comput.*, vol. 24, pp. 51–57, Oct 1991.

[4] M. Ahuja, "FLUSH primitives for asynchronous dist'd systems," *Inform. Processing Lett.*, vol. 34, no. 1, pp. 5–12, Feb. 1990.

[5] P. D. Amer and D. H. New, "Protocol visualization in Estelle," *Comput. Networks and ISDN Syst.*, vol. 25, no. 7, pp. 741–760, Feb. 1993.

[6] G. Andrew, *The Theory of Partitions*. Reading, MA: Addison-Wesley, 1976.

[7] M. D. Atkinson, "The complexity of orders," in *NATO Advanced Study Inst on Algorithms and Order*. Kluwer-Academic, 1989, pp. 195–230.

[8] S. Budkowski and P. Dembinski, "An intro to Estelle: A specification language for distributed systems," *Comput. Networks and ISDN Syst.*, vol 14, no. 1, pp. 3–23, 1987.

[9] G. Brightwell and P. Winkler, "Counting linear extensions is #P-complete," in *Proc 23rd ACM Symp. Theory of Comput.*, pp. 175–181, 1991.

[10] T. Connolly, P. D. Amer, and P. Conrad, "An extension to TCP: Partial order service," (RFC submitted for distribution).

[11] P. Conrad, P. D. Amer, and T. Connolly, "Improving performance in transport layer communications protocols by using partial orders and partial reliability," (submitted for publication).

[12] H. W. Chang, "Linear extensions of partially ordered sets," Tech. Rep. MS Thesis, Carleton Univ., 1986.

[13] T. Camp, P. Kearns, and M. Ahuja, "Proof rules for *FLUSH* channels," *IEEE Trans. Software Eng.*, vol. SE-19, pp. 366–378, Apr. 1993.

[14] M. Diaz and P. Senac, "Time stream petri nets: a model for multimedia streams synchronization," in *Proc. Multimedia Modeling '93*, Singapore, Nov 1993, pp. 257–274.

[15] A. C. Guyton, *Textbook of Medical Physiology*. Philadelphia, PA: Saunders, 1981.

[16] S. L. Hardt-Kornacki and L. A. Ness, "Optimization model for the delivery of interactive multimedia documents," in *Proc. GLOBECOM 91*, Phoenix, AZ, Dec. 1991, pp. 669–673.

[17] Information Processing Systems—Open System Interconnection, *ISO International Standard 9074: Estelle - A Formal Description Technique Based on an Extended State Transition Model.*

[18] D. Knuth, *The Art of Computer Programming Vol 1: Fundamental Algorithms, 2nd ed.*. Reading, MA: Addison-Wesley, 1973.

[19] L. Lamport, "Time, clocks and the ordering of events in a dist'd system," *CACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[20] B. Lampson, N. Lynch, and J. Sogaard-Andersen, "Correctness of at-most-once message delivery protocols," in *Formal Description Techniques, VI* R. Tenney, P. Amer, and U. Uyar, Eds. Amsterdam, The Netherlands: North Holland, 1994.

[21] T. Little and A. Ghafoor, "Network considerations for dist'd multimedia object composition and communication," *IEEE Network Mag.*, pp. 32–49, Nov 1990.

[22] ——, "Synchronization and storage models for multimedia objects," *IEEE J. Select. Areas Commun.*, vol. 8, pp. 413–427, Apr. 1990.

[23] G. Neiger and S. Toueg, "Substituting for real-time and common knowledge in asynchronous dist'd systems," in *Proc. 4th Symp. Principles of Distrib. Comput.*, 1987, pp. 281–293.

[24] L. Peterson, N. Buchholz, and R. Schlighting, "Preserving and using context information in interprocess communication," *ACM Trans. Comput. Syst.*, vol. 7, no. 3, pp. 217–246, Aug. 1989.

[25] I. Rival, *NATO Advanced Study Inst on Algorithms and Order*. New York: Kluwer Academic, 1989.

[26] K. Shafer and M. Ahuja, "Process channel(agent) process model of asynchronous dist'd communication," in *Proc. ICDCS 12*, Yokohama, Japan, June 1992, pp. 4–11.

[27] R. Sijelmassi and B. Strausser, "The PET and DINGO tools for deriving dist'd implementations from Estelle," *Comput. Networks and ISDN Syst.*, vol. 25, no. 7, pp. 841–852, Feb. 1993.

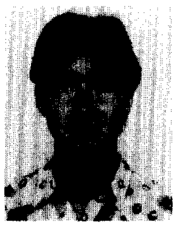
[28] R. Stanley, *Enumerative Combinatorics: Volume 1*. Wadsworth + Brooks/Cole Advanced Books & Software, Monterey, CA, 1986.

[29] J. Valdes, R. Tarjan, and E. Lawler, "The recognition of series parallel digraphs," *SIAM J. Comput.*, vol. 11, no.2, pp. 298–313, 1982.

Paul D. Amer (A'92), received the B.S. degree *summa cum laude* in mathematics from the State University of New York at Albany in 1974, and the M.S. and Ph.D. degrees in computer and information science from The Ohio State University in 1976 and 1979, respectively.

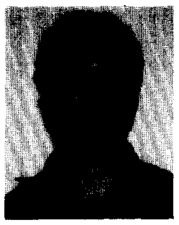
Since 1979, he has been with the University of Delaware where currently he is professor of computer science. From 1978 to 1987, he was employed permanent part-time in Washington, DC, as a Research Computer Scientist at the National Bureau of Standards. In 1985-1986, he was at the Agence de l'Informatique in Paris contributing to the development of Estelle, now ISO International Standard 9074. In 1992-1993, he was at the Laboratoire d'Automatique et d'Analyse des Systemes (LAAS) of the Centre National de la Recherche Scientifique (CNRS) in Toulouse, France working on a partial order transport service. His current research interests are: protocols for high speed networks, formal specifications of ISO protocols and services; protocol visualization of specifications; automatic protocol test case generation; and extensions to Estelle.

Dr. Amer is on the Editorial Boards of *Computer Networks and ISDN Systems* and *Reseaux et Informatique Repartie*. He has also been a member of the ACM since 1976.



Christophe Chassot received the Diplome d'Ingenieur and DEA in Computer Science from the ENSEEIHT of Toulouse in 1992.

Currently he is working at LAAS (Laboratoire d'Analyse et d'Architecture des Systemes du CNRS, Toulouse) on the Ph.D. degree in computer science from the Institut National Polytechnique de Toulouse. His main fields of interest include multimedia transport service and protocol formal specification.



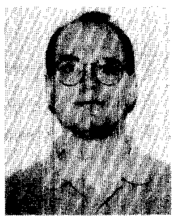
Thomas J. Connolly (S'86) received the B.S. degree in electrical engineering from the University of Notre Dame, South Bend, IN, in 1987, and the M.S. degree in computer engineering from Villanova University, Villanova, PA, in 1990. He is currently a candidate for the Ph.D. degree in computer science at the University of Delaware.

Since 1987 he has been involved with computer and network design. His current research interests include the design and analysis of protocols for distributed systems.

Mr. Connolly is has been a member of the ACM since 1992.

Michel Diaz (SM'92) received the Doctorat es Sciences in 1969 from the University of Toulouse. He is a Directeur de Recherche at the Centre National de la Recherche Scientifique (CNRS) and leads the Research Group "Communications Softwares and Tools" at Laboratoire d'Automatique et d'Analyse des Systemes du C.N.R.S., Toulouse. He has been working on the development of formal methodologies, techniques and tools for designing distributed systems during the last ten years. From 1984 to 1988, he was manager of the SEDOS project (Software Environments for the Design of Open distributed Systems, in which the Formal Techniques Estelle and LOTOS have been developed) within the ESPRIT program of the ECC. In 1989-1990, he spent a year as a visiting staff at the Universities of Delaware and California at Berkeley.

Dr. Diaz is a member of many program committees, he served as a Program Chairman for the IFIP Conference on "Protocol Specification, Testing and Verification," the European Workshop on "Application and Theory of Petri nets," and the International Conference on Distributed Computing Systems, in Area "Software Engineering," the IFIP conference on Formal Description Techniques. He a Technical Editor for *Reseaux et Informatique Repartie*, *Annales des Telecommunications* and *Communications Magazine*. He has written one book and more than 100 technical publications. He is the editor of the North Holland volume on Protocol Specification, Testing and Verification, 1985, co-editor of two North Holland volumes dedicated respectively to the Formal Description Techniques Estelle and LOTOS, 1990, and co-editor of the North Holland volume on Formal Description Techniques, 1992. He is presently Director of the French Research Coordination Group on "Parallelism, Networks and Systems" (GDR Parallelisme, Reseaux et Systemes) and the co-head of the French CNET-CNRS collaborative project CESAME on the formal design of high speed multimedia cooperative systems.



Phillip Conrad (S'94) received the B.S. degree in computer science from West Virginia Wesleyan College, and the M.S. degree in computer science from West Virginia University. He is currently working towards the Ph.D. degree in computer science at the University of Delaware.

His research interests include networking and algorithms. His email address is: pconrad@cis.udel.edu.

Mr. Conrad has been a member of the ACM since 1992.