

COORDINATED TRANSMISSION OF DATA IN A DISTRIBUTED MULTIMEDIA  
CONFERENCING APPLICATION

BY

DENNIS JAY FOREMAN

BS, Youngstown University, 1967  
MS, Binghamton University, 1974

DISSERTATION

Submitted in partial fulfillment of the requirements for the  
degree of Doctor of Philosophy in Computer Science  
in the Graduate School of  
Binghamton University  
State University of New York  
2002

© Copyright by Dennis Jay Foreman 2002.  
All rights reserved.

Accepted in partial fulfillment of the requirements for  
the degree of Doctor of Philosophy in Computer Science  
in the Graduate School of  
Binghamton University  
State University of New York  
2002

Dr. Walter Piotrowski \_\_\_\_\_ April 2, 2002  
Computer Science Department

Dr. Kanad Ghose \_\_\_\_\_ April 2, 2002  
Computer Science Department

Dr. Michal Cutler \_\_\_\_\_ April 2, 2002  
Computer Science Department

Dr. James Constable \_\_\_\_\_ April 2, 2002  
Electrical Engineering Department

## **Abstract**

This thesis examines the ability to coordinate the transmission of separate data streams (possibly consisting of graphics, sound, text, and/or video) in a multimedia group application. Most of the work that has been done with respect to managing distributed multimedia data streams is in the area of guaranteeing rates of presentation over a network and has been with respect to a single source and/or a single destination. This thesis describes a program that provides a form of multi-cast transmission for independent streams of data that must be presented together at their destinations. We give the name *Ensembles* to these collections of streams of data. The program provides coordination of Ensembles for multiple senders and receivers in multiple groups without referring to a global time mechanism. The coordination is accomplished by two mechanisms: specification of a header attached to each transmitted segment, that carries with it the distribution information for that segment, and an algorithm for using the header with low computational overhead and minimal buffering.

## **Dedication**

To Sandy, my wife, who put up with the clutter and kept me focused. To my parents, who encouraged my curiosity and the desire to study.

## Acknowledgements

I wish to offer my thanks to several people who offered advice, ideas, and encouragement during the development of this thesis and the research leading up to it: Dr. Joe Cornachio, who was my advisor until his retirement and who was the first to plant the seed of attempting this journey; the IBM Corporation, which provided the initial laboratory equipment and continued support thereof for many years; the members of the IBM Endicott VM Development organization who had faith in my work and continued to offer their assistance across the 'downsizing' years; Dr. Kanad Ghose, who provided much encouragement and the suggestion of using LEDA© to provide a multi-platform C++ environment; Dr. Jose Delgado-Frias, who made several useful suggestions about related work in computer architecture, especially the work on *flits* and the possibilities exhibited in the Wormhole Routing papers; Michael Seel, LEDA development, for his assistance with the use of LEDA and the platform independent mechanism in LEDA for working with graphs and windows; Marco Nissen, LEDA development, who provided me with valuable insights into the graph manipulations in LEDA, and gave me his FindIt program for graph searching; Dr. Walt Piotrowski, who encouraged me, helped me through the TCPIP maze; and provided valuable suggestions on the best way to put forth my discussions; the technical support staff at Binghamton University: Christine Place-Sweet, John Hagan, Donald Kunkel, Victor Fiori, and Robert Mess for their hints, tips and systems support experience; and last, but not least, Dave Butenhof, Ross Johnson and John Bossom for their work on Pthreads for Windows, which made the multi-platform thread coding significantly less painful than it would have been.

I would also like to thank Dr. Michal Cutler, Dr. Richard Fischer, Mark and Dr. Terry Dylewski and Deanna France for reviewing this thesis and their suggestions for improvement.

# Contents

1. Introduction.....	1
2. Terminology.....	5
2.1. Basic Terms .....	5
2.2. Ordering Properties.....	7
2.3. Synchronization Types.....	9
2.4. Terms Developed in this Thesis.....	12
2.4.1. Ensemble.....	12
2.4.2. Ensemble Rule .....	13
2.4.3. Ensemble Set.....	14
2.4.4. Ensemble Coordination.....	16
2.4.5. Intermediate Synchronization Server.....	16
3. Dissertation Topic.....	17
3.1. Contribution .....	17
3.2. Basis.....	18
3.3. Problem Characterization.....	22
3.4. Development.....	23
3.4.1. Issues.....	26
3.5. Objectives .....	28
3.6. Approach to the Solution .....	29
4. Review of Literature and Prior Related Work .....	34
5. System Architecture.....	41
5.1. Notation Conventions .....	41
5.2. Network Conventions .....	41
5.3. Restrictions .....	42
5.4. Security .....	43
5.5. Multi-platform Considerations.....	43
5.5.1. Multi-tasking.....	43
5.5.2. Programming Languages .....	47
5.5.3. Network Interfaces.....	48
5.5.4. Error Handling .....	48
5.6. Distributed Execution .....	49
5.7. Basic Program Description .....	49
5.8. Implementation Stage 1 .....	52
5.8.1. Initialization .....	53
5.8.1.1. Build_graph.....	54
5.8.1.2. Build_E.....	54
5.8.1.3. Start_E_window_mgr .....	56

5.8.1.4. E_window_mgr.....	56
5.8.1.5. Start_Receiver.....	56
5.8.1.6. Select_thread.....	57
5.8.2. Preparation for Stage 2.....	58
5.9. Implementation of Stage 2.....	59
5.9.1. Sending Data.....	62
5.9.1.1. Q_E_Files .....	62
5.9.1.2. Send_Ensembles .....	64
5.9.1.3. Xmit_data.....	64
5.9.1.4. Out_Qmgr .....	65
5.9.1.5. Sndthread .....	65
5.9.2. Receiving data.....	66
5.9.2.1. Rcvthread .....	66
5.9.2.2. In_Qmgr.....	66
5.9.2.3. Q_E_Net .....	67
5.9.3. Multi-casting in the ECP.....	68
5.10. Data structures Classes & Methods .....	69
5.10.1. The Graph Node Descriptor ( <i>Gnodedata</i> ) .....	69
5.10.2. The Ensemble Object ( <i>E_obj</i> ).....	70
5.10.3. The Stream Object ( <i>S_obj</i> ).....	71
5.10.4. The Stream Object List ( <i>S_objlist</i> ) .....	71
5.10.5. The Stream Route Object ( <i>S_route</i> ).....	71
5.10.6. The Path List .....	72
5.10.7. The Pathnode Vector ( <i>PV</i> ) .....	73
5.10.8. Distribution Vector ( <i>DV</i> ) .....	73
5.10.9. The Send/Receive Packet ( <i>packet_def</i> ).....	74
6. Inputs and Outputs .....	75
6.1. The Node and Ensemble Definition File ( <i>ne.dat</i> ) .....	75
6.2. The host locator file ( <i>node2ip.dat</i> ).....	76
6.3. The Ensemble Rule Descriptor File ( <i>erules.dat</i> ).....	76
6.4. Input Data ( <i>s1.dat</i> through <i>s5.dat</i> ).....	77
6.5. Output Data.....	78
6.6. Output of E1 at nodes D1 and D3.....	78
7. The Coordination Algorithm.....	81
8. Testing, Analysis and Results.....	83
8.1. Testing.....	83
8.1.1. Initialization .....	83
8.1.2. Running the Program .....	85
8.1.3. Measurements .....	88
8.2. Analyzing the Ensemble timing data .....	89
8.3. Results.....	90
9. Conclusions.....	92

10. Future Work .....	94
11. Bibliography .....	95

## Figures

Figure 1. Ordering Properties .....	9
Figure 2. Synchronization Types .....	10
Figure 3. An Ensemble .....	12
Figure 4. The content of a conference .....	14
Figure 5. A graphical view of a conference.....	15
Figure 6. A multi-origin, multi-destination conference.....	27
Figure 7. Program Architecture, Stage 1 .....	53
Figure 8. Program Architecture, Stage 2 .....	61
Figure 9. The Graphnode & E_obj Descriptor Structures .....	70
Figure 10. Adjacency Matrix for the directed graph, G.....	72
Figure 11. Output Recorded for E1 at both D1 and D3 .....	80
Figure 12. Recorded Timing Data for E1 .....	90

## 1. Introduction

The ability to communicate using live video and audio transmission over computer networks has given rise to the concept of multimedia computer conferencing. A major requirement in multimedia conferencing is the ability to have simultaneous transmission of multiple streams of related data and to allow for more than one arrangement of origins and destinations in a conference. Since most computers have only one physical network connection, only one piece of data can actually be transmitted at any moment. Because of the potential for complexity caused by multiple sources of transmission and/or multiple destinations, there is a need to manage these transmissions tightly and with speed. In many of these situations, the streams must arrive at each of the destinations in a coordinated manner; that is, some pieces of data may be required to be displayed together at their destination, in some time-related manner, relative to their original transmission. Some examples of conferences that exhibit this complexity are:

- A. A physician, at location A, must interpret a fluoroscopic film of a patient. The physician at A wants to consult with physicians at locations B and C, who are experts in this type of case. While viewing the film, the physicians will discuss it. They may: stop and review parts of the film; annotate the film with audio comments, picture-in-picture video, pen markings, or text; record their conference to be part of the patient's medical file; and/or view other pertinent documents such as previous fluoroscopic results, X-rays, or lab reports. The physicians remain in their respective offices during the consultation.

- B. An on-line business meeting, where executives send messages, voice and images to one another, with an (essentially) unlimited number of attendees. The attendees act and react to the content of each other's messages, in near real-time fashion.
- C. A symphony, where the participants send their voices, music and images to each other. All performers can respond to each other's transmissions in close to real-time fashion.
- D. A collection of aircraft simulators, where it is desired that signals in each simulator are shared, giving the effect of performing together, though (possibly) physically separated by great distances. Or a variation where a group of simulators must receive multiple sensor information in a coordinated manner.

There are drawbacks with all current methods of accomplishing these transmissions, such as:

- A. Large buffers may be required for storing the data at the receiving locations imposing size and cost constraints on all participants.
- B. A centralized server may be required, which receives all data and re-transmits it in a coordinated manner, thus becoming a bottleneck for the conference.
- C. Some solutions require specific response times, that must be managed by all nodes in the network that handle the data. A network modification called Quality of Service

management has been used in some cases to provide these response times and has been extensively studied in other papers. It involves additional overhead on every system between source and destination, even if the system is not actually involved in the conference.

- D. Timing algorithms (such as the Byzantine Generals protocol) are sometimes used to guarantee the synchronous delivery of data. Several such protocols have been used, with varying degrees of success, but management of the protocols involves significant system overhead, added network traffic and added complexity in the application.
- E. Some solutions require foreknowledge of the specific content of all datastreams (video vs audio) and their timing inter-relationships. Synchronization is then only done in the media application. This method may rely on one or more of the preceding drawbacks for additional support. Multiplexing the data before transmission is an example of a solution that requires foreknowledge of the stream content. <sup>1</sup>

The *Ensembles Coordination Program* (ECP), described in this thesis, is an attempt to resolve these drawbacks. The ECP was created to provide the desired coordination of multi-stream data, for multiple source and multiple destination conferences and includes the situation where more than one collection of streams may appear at any destination. The ECP runs as an application program in UNIX and Windows. The ECP coordinates the transmission and reception of multimedia data

---

<sup>1</sup> To date, this has only been attempted when all data streams originate at the same location.

stream segments, permitting the content to be processed (displayed) by a user application. For simplicity in testing, the ECP contains a simulated user application for display of the streams.

This thesis documents the ECP and presents the results of the implementation. A brief review of the relevant terminology (Chapter 2) provides the foundation for further discussion. Two of the terms described in that chapter, *Ensemble* and *Ensemble Set*, describe the elements of the conference data that are controlled by the ECP. Another term, *Ensemble Rule*, describes the inter-relationships among the streams.

The balance of this thesis consists of the dissertation topic, including the significance of the work, further characterization of the problem, the issues involved in the topic and the approach taken to reach the solution. The objectives are then described and a review of the current literature and related work is discussed. The implementation details are then given followed by the results of the implementation. A description of future work is provided, followed by a section containing key portions of the actual source code.

## **2. Terminology**

Before proceeding, the terms I have used must be understood, some of which are in common use in the area of multimedia investigations and others that are used here for the first time. Following the basic terms are the Ordering Properties and Synchronization Types for messages transmitted and received in a distributed multimedia environment. These terms are at the heart of understanding the problem of managing multiple data streams. I have attempted to discuss them in detail after the basic terms are described. These are followed by the terms used in my thesis.

### **2.1. Basic Terms**

A *stream* is the sequential arrangement of the elements of a single continuous medium, such as an audio or video file. By continuous, I mean that the source continues to transmit (possibly empty) segments of data, even when the source is inactive. This would be the case when a video camera is turned engaged, but there is only a background being transmitted. Likewise, when a microphone is turned on, it continuously transmits the background sounds of the room it is in. It might be possible for the conference application to compress this background information before transmitting it. This would be transparent to the operation of the ECP.



example of an activity of a video object that is being presented. There may be a multiple operations (methods) defined for each multimedia object. The multimedia activity is the currently active operation.

A *group* is a collection of participants, actively or passively involved in a conference. Members of a group send and/or receive a specific portion of all the available streams in the conference. A participant may be in more than one group and a conference may contain more than one group. Data streams used in one group may also be used by another group.

In collaborative multimedia applications, the *object model* represents the multimedia data streams and their operations. A “multimedia object” is composed of “activities” and “multimedia streams” [Ste90]. A multimedia stream corresponds to “the data specific for one or more media.” If an audio and video stream are transmitted as an object, there is no mechanism automatically associated with the object for specifying how the two streams are to be played back other than to start them both at the same time and play them to their conclusion. A stream specifies only that the data are related and in sequence, but does not necessarily require individual segments of data to be presented together in real-time synchronization.

## **2.2. Ordering Properties**

Garcia-Molina[GS91] has described three *Ordering Properties* and has defined these relationships for multicast messages in a distributed system. I have created Figure 1 on page 9 to illustrate these relationships. Please note that there is no mention of the time between messages, as

the properties refer only to the order of transmission and receipt. The three *Ordering Properties* are:

1. Single source ordering - This requirement states that if messages  $m_1$ , and  $m_2$  are sequentially generated at the same site and within a specific multicast group, they are received in the same order by all receivers in the group.
2. Multiple source ordering – This requirement states that if messages  $m_1$  and  $m_2$  are sequentially generated by *different* sources, they will arrive at all recipients in the same group and in the same order as they were generated. There is no specification of managing a time differential between  $m_1$  and  $m_2$  in the receipt of the messages.
3. Multiple group ordering - This requirement states that sequentially generated messages  $m_1$  and  $m_2$ , delivered to two processes (independent programs on one or more computers) will be delivered in the same order, even if they originated at different sources, regardless of the receivers' multicast group memberships. In other words, group membership does not affect ordering of the receipt of the messages.

Ordering Property 2 is called an Inter-stream Ordering Property, because it specifies the ordering of the segments of multiple streams with respect to all of the streams, as if a global timestamp were possible.



- II. *Inter-object synchronization* refers to the synchronization between media objects at a single site. An example of this is the merging of audio and video in a movie, such that the speaker's lips move at the same time that the words are spoken.

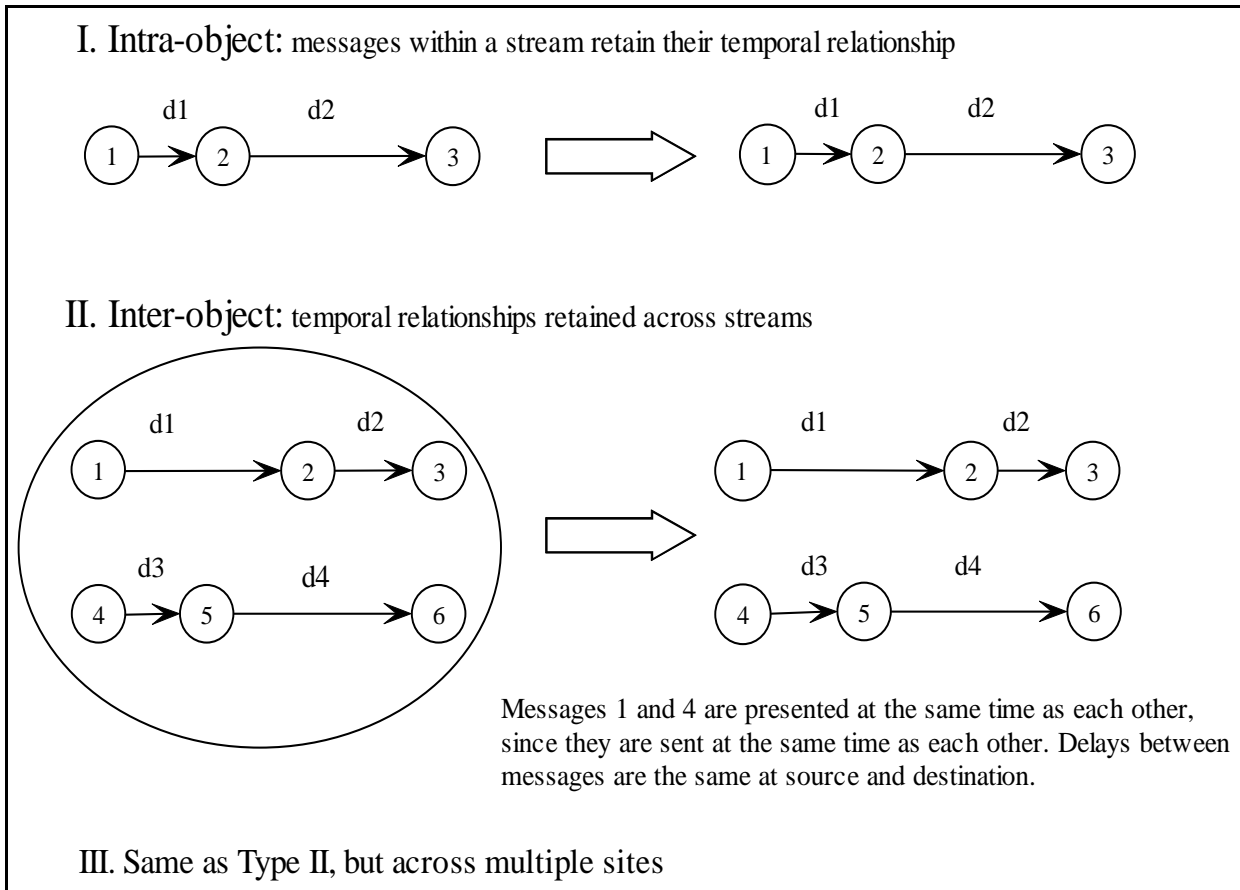


Figure 2. Synchronization Types

Blakowski also mentions that there are problems (without delineating them) involved in the output of synchronized streams at multiple sites. These problems are part of a third problem, which I call Type III, or *Site Synchronization* (which might be referred to as synchronization within a multicast group). Site synchronization refers to the synchronization of Inter-object presentation of the active objects at different nodes in the conference such that all sites display

the same objects with identical synchronization and identical delays within and between the objects. Site Synchronization does not require that multiple destinations display their output in any specific time-relationship across sites (with regard to position within media).

One might also consider a fourth type of synchronization, (IV) Global Synchronization, wherein all sites provide Synchronization Types I and II as well as Type III in a **close-to-universal-time** presentation of object sequences, as in a live television broadcast. This fourth type of synchronization is a possible subject for future work.

*Intra-stream (or Intra-object) Synchronization* guarantees that the temporal correspondence between media units **within** a single medium will be preserved at presentation [JSRM95]<sup>2</sup>.

*Inter-stream (or Inter-object) Synchronization* guarantees that the temporal links **across** streams of data will be preserved at presentation [CFC\*96].

The basic difference between Synchronization and Ordering Properties is that Synchronization specifically addresses the temporal relationships of the streams' contents and is defined for multiple *destinations*, without regard to group memberships or the origins of the streams. The Ordering Properties specifically address the issue of multiple *origins* as well as multiple *destinations*, in terms of delivery of messages and group membership, but do not address the temporal relationships of the streams' contents.

---

<sup>2</sup> Although most of the work in that paper is focused on video 'jitter', the discussion of Intra-stream Synchronization is worth noting.

## 2.4. Terms Developed in this Thesis

### 2.4.1. Ensemble

An *Ensemble* is the smallest collection of segments of all datastreams that must be presented together at one or more destinations. An Ensemble is not required to originate at a single location, but must be available at each of its destinations before presentation of the data can begin. We can think of an Ensemble as a molecule of information, the smallest collection of segments of all streams that provides a coherent output from all the streams. An example of an Ensemble is seen in Figure 3.

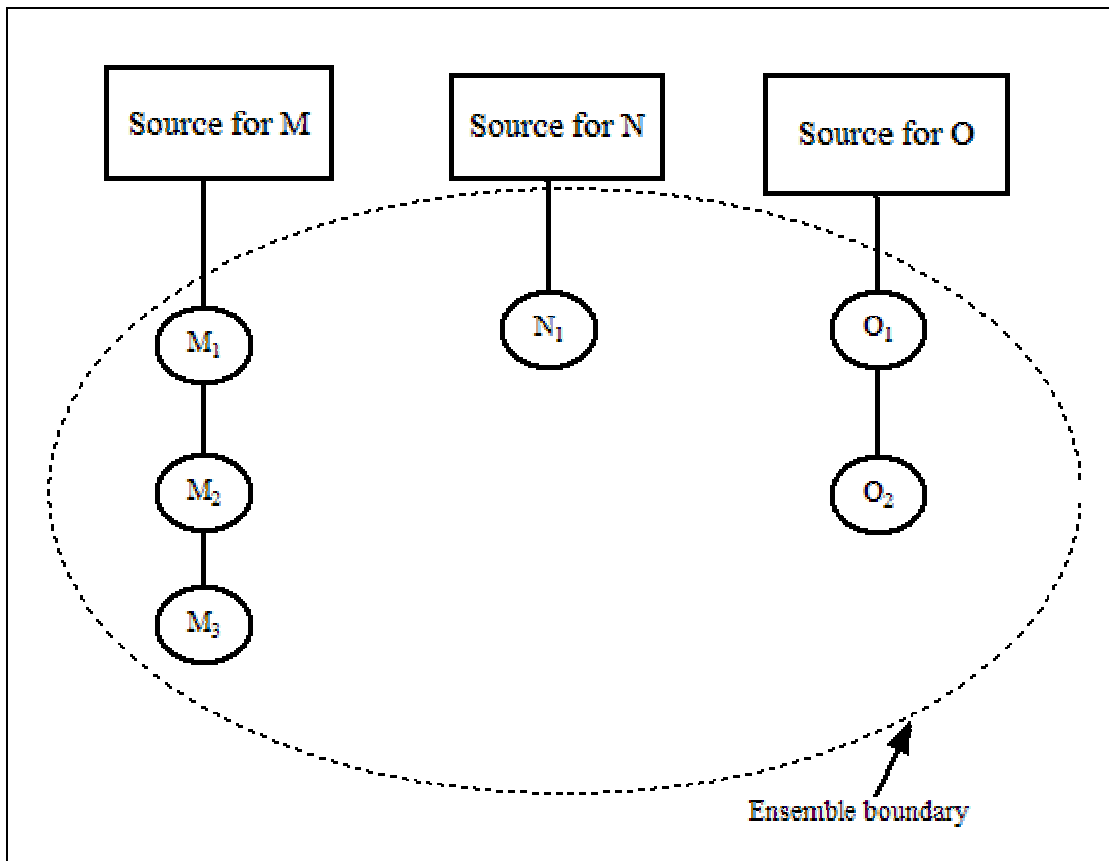


Figure 3. An Ensemble

## 2.4.2. Ensemble Rule

The coordination of data within a conference is based on a set of *Ensemble Rules* that describe the inter-relationships among the various data streams within the conference. An Ensemble Rule is related only to the media streams being used and is independent of the content of the streams. The streams are related by the rates of presentation of their media.

In order to have enough information for one presentation moment (or molecule), we must have all the requisite segments of each stream present at each destination. For example, consider the conference in Figure 4 consisting of video, audio and text data, coming from three sources and being presented at these same three locations as destinations. The information for one digital video frame (stream S1 from source A) may require 3 segments for transmission, whereas one audio syllable (stream S2 from A) may need only 2 data segment for transmission whereas the analog video (stream S3 from B) may require 4 segments for transmission. The audio from B (stream S4) will require 2 segments and the text stream from C will only require 1. Together, these 11 transmission segments are an Ensemble, E1, which is transmitted (in this example, from a central location), to each destination (A, B and C). In this example the ratio of elements for the Ensemble is fixed at: S1 - 3 segments, S2 - 2 segments, S3 - 4 segments, S4 - 2 segments, and S5 - 1 segment. This ratio is the Ensemble Rule for this Ensemble of the conference in Figure 4. In section 6.3 on page 76, we show how these ratios are encoded for input to the ECP. A conference may have more than one Ensemble.

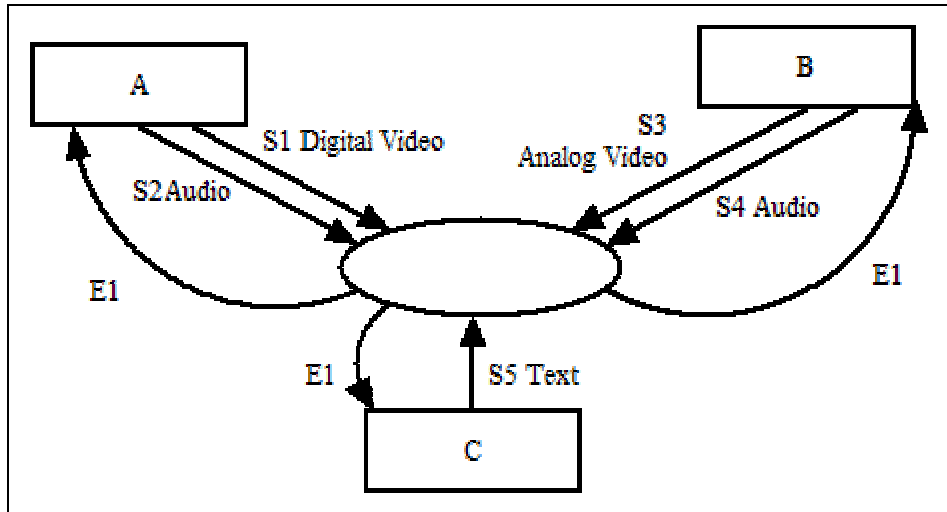


Figure 4. The content of a conference

### 2.4.3. Ensemble Set

An *Ensemble Set* is the collection of streams of data that require adherence to Ordering Properties 1, 2 and 3. When an Ensemble Set is to be transmitted, the data is sent from each stream origin in segments, rather than as a continuous stream. The segments are then gathered by the ECP into Ensembles and re-transmitted toward their destination. It is important to note that a stream may be used in more than one Ensemble Set, thus requiring the ECP to track the segments of each Ensemble and re-use them as required.

An example of an Ensemble Set is the audio and video portions of a digitally recorded movie having a voice synchronized with the lip movement of the speaker, wherein the audio and video streams of data are (possibly) sent on separate data channels. In addition, there may be more than one data segment of video information for each audio segment and their sizes may be different. An Ensemble for this Ensemble Set might be the segments of audio and video for  $1/30^{\text{th}}$  sec (a frame) of

the movie. “Maintaining *lip-sync* between audio and video data in video-on-demand applications is an example of inter-stream synchronization requirement” (sic) [AWG94].

A graphical example of a conference is seen in Figure 5. The Ensemble Set for this figure consists of the streams S1, S2 and S3 displayed for all members of the group (D1 and D3).

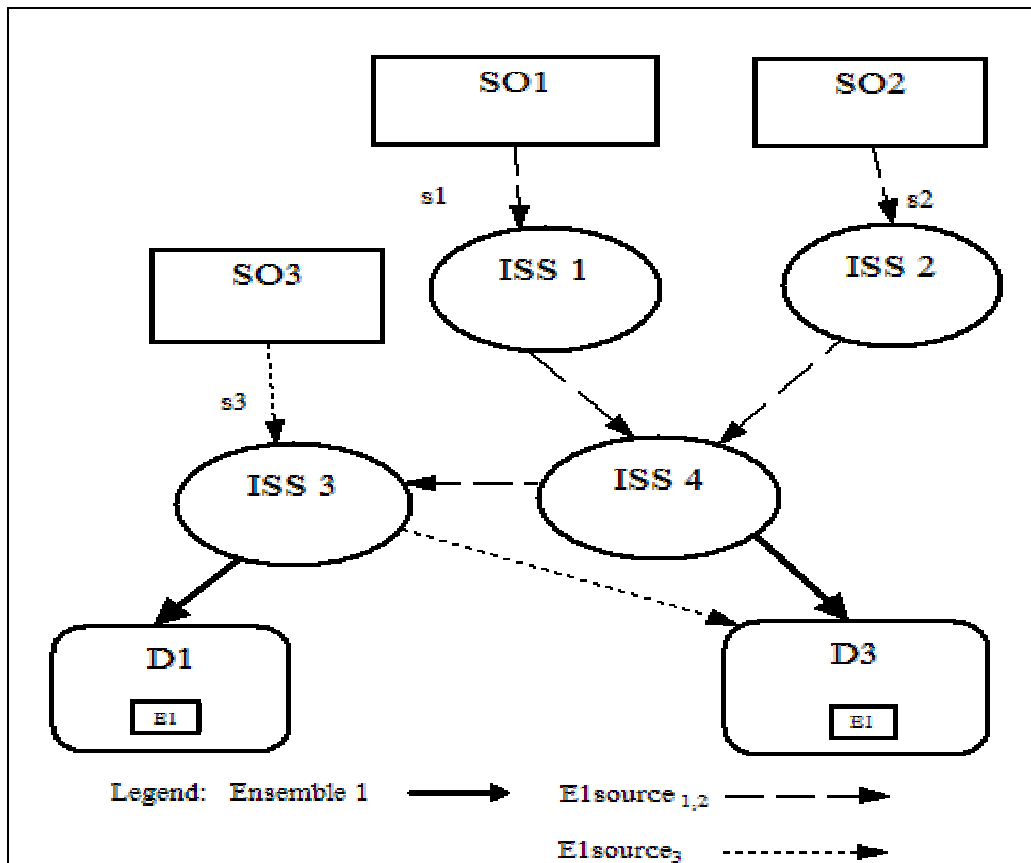


Figure 5. A graphical view of a conference

#### 2.4.4. Ensemble Coordination

*Ensemble Coordination*, the purpose of the ECP, is the process of ensuring the cohesive transmission of elements of an Ensemble Set, with minimal requirements for buffering. That is, Ensemble Coordination is a process that provides Ordering Properties 1, 2 and 3.

The *Ensemble object model* includes:

1. the definition of the number of streams,
2. the inter-relationships of the streams and
3. the specification of the required number of segments of each stream all of which must arrive at a final destination in the set of streams being managed.

Item 3 does not imply that the ECP has knowledge of the content type (audio, video) of the stream.

Since the design of the ECP is oriented toward data flowing through a network, there is no effort made to manage persistent data.

#### 2.4.5. Intermediate Synchronization Server

An *Intermediate Synchronization Server (ISS)* is a service node in the conference network that gathers segments of streams to form Ensembles. At an ISS, it may not be possible to form a complete Ensemble, because it may be the case that all the streams needed for that Ensemble do not flow through that node. It is therefore the responsibility of the ISS to gather those stream segments that it can control and forward as much of the Ensemble as it is possible to gather, to the next node in the path to the destination.

### **3. Dissertation Topic**

This thesis examines the technical issues of a distributed multimedia group application using a form of multicasting to transmit information across the network. I have constructed a program that utilizes an algorithm for its multicasting operations that reduces the effects of delays and provides for a closer-in-universal-time presentation of the data. I have done this without using any fixed universal time reference or time stamp algorithms.

#### **3.1. Contribution**

To date, to the best of my knowledge, the problem of coordinating streams of data has been approached by a combination of one or more of the following techniques: Quality of Service (QoS) management, buffering large quantities of data, pre-transmitting data before playback and/or creating cohesive data streams (video and audio in a single file or stream) via some form of multiplexing. In order to appreciate the contribution of this thesis, the drawbacks of these techniques must be understood.

Quality of Service management requires a knowledgeable network administrator to establish the QoS measures. In addition, the QoS measures must be maintained at every node through which the data must pass. QoS management adds complexity to the network programs and contributes a delay to the processing of all network traffic.

Buffering often requires large quantities of storage, which may not be available at all nodes, thus tying up resources until space becomes available. For Multimedia applications, this could require hundreds of megabytes per stream, which adds up to a significant amount of space that must be reserved for the conference.

Pre-transmitting data precludes the possibilities of live transmissions because the receiver must wait for all data to be transmitted before viewing any of it, although recent advances in streaming audio and vi

the conference. Also, there is only one instance of the conference (the conference is neither duplicated nor reproduced anywhere).

In a multimedia group conference, where any conference member may be in multiple discussion-groups simultaneously and multiple streams of data are being presented, the streams must maintain their original ordering. In an ECP managed conference, this means that one ECP computer may be sending and/or receiving multiple collections of coordinated data, each appearing in its own window, requiring coordination of its streams of data independent of the coordination in other windows.

As an example of a traditional conference, consider a group of people gathered in a room to discuss a scene on a videotape. Except for the minor differences in distance covered at the speed of light within the room, every participant sees the same frames at the same time. Also, every participant can see every other participant at the same time.

In a distributed conference, there is no direct connection between participants, therefore the streams of data cannot be displayed continuously, but data must be transmitted over a network in relatively small packages. Each package is surrounded by the information that permits the network transmission facilities to properly send the package to its destination. Each such package, including the surrounding control information, is known as a network packet. There are many methods for transmission of such packets, such as Asynchronous Transfer Method (ATM) and Transmission Control Protocol/Internet Protocol (TCP/IP) but each methodology has its own inherent causes of delay. Because of these delays, regardless of the mechanism chosen, each participant will see the

different elements of the conference data at different universal times. Therefore, although there will be delays, data that is to be "played" together must continue to have those intra- and inter-stream relationships maintained.

In a distributed conference, participants are at different physical locations and sufficiently distant from one another that electronic transmission delays become important. Electronic transmission delays include the time spent in the transmission media (wire, fiber optic cable, radio waves, etc) and the delays caused by the routing software within the sending and receiving nodes, as well as within any nodes between them.

If a global clock existed, all transmitted data could be time-stamped and both ordering and synchronization could be performed by the multimedia application, without regard to the origin of the data streams. However, since there is no global clock, the only mechanisms to ensure the ordering and synchronization of data involves some type of buffering facility with coordination algorithms to manage the differing streams of data and their content.

A participant in a distributed multimedia conference uses a computer that is a node in a computer network. Networks may be interconnected. A communications packet may be retransmitted many times before it arrives at its final destination, because the sender and receiver are not directly connected. In addition, a given packet may be required by many conference participants, and will therefore have to be transmitted to all of them. This will require additional re-transmissions. The operation of transmitting a single packet to multiple specific recipients is known as multi-casting. This differs from broadcasting. In broadcasting, there is no attempt to specify any particular

destination; packets are made available to any computer on the network that is prepared to receive them. In traditional multi-casting, the application has no control over the ordering mechanism for data being sent to multiple destinations on the network. In a conference using the ECP, multi-casting is managed by the ECP, allowing for direct control over which data packets are sent, where they are sent and in what order they are sent.

When we consider a distributed conference, as we do in this thesis, we need to distinguish between “doing” and “seeing.” When a conference participant **does** something (e.g. moves a mouse), the feedback on his own (and other) systems may not be instantaneous. All participants, including the participant performing the action, will **see** the movement of the mouse pointer in the same way; ordered with respect to other datastreams by the three Ordering Properties noted earlier, but they may not see it at the same time as it occurs. Note that, since there is no common global time, even when the Ordering Properties are fulfilled, all participants see events in the same order but not necessarily at the same *universal* time.

On a very fast LAN, the human perception of the presentation of the data at a receiver will be nearly instantaneous with the transmission. With other network connections there may be a noticeable delay between doing and seeing. For some types of conferences and datastreams, some amount of delay may be acceptable; for others, the same delay may tax the adaptability of the participants or become completely disorienting. The potential human effects of delay are interesting to contemplate but they are not the subject of this thesis. Neither are the nature, causes and corrections of the delays within the network. We leave exploring and quantifying these extremely interesting issues for future work.

The nature, causes and corrections of the delays within the network caused by these protocols are not the subject of this research. In addition, because there is no universal time, there is no way to determine if all participants are actually seeing the same set of packets at the same time.

I have already mentioned that one possible method of managing the synchronization requirements is to multiplex the signals but Blakowski [BS96] acknowledges the difficulty in multiplexing data streams. He states that it is a matter of “selecting a Quality of Service level that matches the requirements of all involved medias...” (sic) and later mentions that multiplexing is difficult to use for multiple source nodes.

Since we consider multiplexing a function of the application that requires foreknowledge of the content of every datastream and since it is not a requirement for mixed-type data streams, we do not use it.

### **3.3. Problem Characterization**

For convenience, I have noted and summarized 3 important areas covered by this thesis:

1. The basic problem in multimedia conferencing systems is the ability of such a system to provide at least the first two Synchronization Types and the three Ordering Properties (previously discussed in 2.2 on page 7). There have been several approaches to this problem, some of which rely upon the operating system and the network to provide a Quality of Service for transmission, with buffering of data by the receiver or a central sender. My thesis explores the problem of providing coordinated transmission of multiple

streams of time-related data, so that a sufficient number of segments of all streams (an Ensemble) are available at each receiver for playback, while at the same time, reducing the extensive buffering required to ‘play’ the data with ‘lip synch’.

2. Packet management overhead, such as time-stamping and QoS, adds overhead to every node in the conference network, regardless of whether that node is processing coordinated data or not. The ECP only adds overhead to nodes that process Ensembles.
3. There is no global clock. Therefore, transmitted data cannot be time-stamped. This complicates methods for providing both Ordering and Synchronization by the multimedia application or any support mechanism, regardless of the origin of the data streams. Since there is no global clock, some kind of buffering facility with coordination and synchronization algorithms is needed to manage the differing streams of data and their content.

### **3.4. *Development***

A conference is described as the presentation and reception of streams of information. A multimedia conference consists of the simultaneous (on one computer) presentation of multiple streams of sequentially related information by an application program. These streams are the sounds and images of audio and video information, as well as text and graphic data being presented. In such transmissions, there is a sequential relationship within a single stream. For example, the

sounds of spoken words must be reproduced at the destination, in the same order in which they were spoken or the meaning is lost. This is an example of Type I (Intra-object) Synchronization.

There is also a sequential dependency between streams. For example, the voice of a person speaking must coincide with the movement of the person's lips on the screen. This is an example of Type II (inter-object) Synchronization. Thus, during any conference instant, the elements of one stream must be synchronized with the elements of all other streams related to that presentation-instant, (corresponding to a single movie frame).

Let us assume an environment with several streams, M, N, O, etc., where each stream, for example M, consists of multiple segments of sequential data,  $M_1, 2, 3$ , etc. In actual usage, the nature of the data in each stream might require that the segments of these streams do not match each other, one by one. For example, a video plus audio transmission with graphical data might consist of multiple elements,  $x$ , of video data (stream M) for one element,  $y$ , of audio information (stream N), considering that video data consists of 30 pictures per second, and some number of elements,  $z$ , of the graphical data stream (stream N). In situations like this, we need to arrange for a different number of segments from each of the streams to be available together at their destination. We would therefore like to have every  $x$  elements of M, every  $y$  elements of N and every  $z$  elements of O to be transmitted together where  $x$ ,  $y$  and  $z$  have a defined relationship. By doing this, we guarantee that all of the data for a presentation instant will be available together.

In order to maintain the sequential relationships within each stream and across multiple streams, a mechanism is required to ensure that the  $i^{\text{th}}$  group of  $x$  elements of stream M are presented to the

application with the  $i^{\text{th}}$  group of  $y$  elements of stream  $N$  and the  $i^{\text{th}}$  group of  $z$  elements of stream  $O$ . It is also possible that the streams could have a ratio of segment transmission where the segments of streams  $M$ ,  $N$  and  $O$  might not be transmitted at a fixed rate. Rather, there is a (possibly non-linear) relationship between the values of  $x$ ,  $y$  and  $z$ . We therefore developed the expression,

$$E_{M,N,O\dots}(x, y, z, \dots)$$

that relates the presentation requirements of streams  $M$ ,  $N$  and  $O$  at multiple destinations in a network. In the example previously described in Section 2.4.1 on page 12 where we defined an Ensemble, we have  $x=3$ ,  $y=1$  and  $z=2$ .

The concept of Inter-stream Coordination may be extended to more than 3 sources, as when the individual sensors on a battlefield separately send their data to multiple recipients or when multiple members of a distributed symphony play their individual parts, all expected to be played at their destination in correct order.

Since the nature of multimedia conferencing is human communication, it is also desirable to have every node,  $\alpha$ , in a conference of  $G$  nodes, display the same collections of data (Ensembles), such that all the human participants of the conference receive the sequential Ensemble groups within some pre-defined offset value of each other,  $\delta$ , as measured at the site where the Ensemble is displayed. The value of  $\delta$  has been the subject of much research on Quality of Service (QOS). My thesis is not concerned with the value of  $\delta$ , other than an attempt to quantify the effects of our use of the expression,  $E$ , on  $\delta$ .

In this thesis, I have concentrated on the problems involved with multi-source, multi-destination coordination of interdependent data streams. These problems are related to maintaining the inter-stream object delivery, which leaves the temporal relationships to some other application that is aware of those relationships.

### 3.4.1. Issues

The subjects of Intra-object Synchronization and Inter-object Synchronization have been discussed by many researchers, including Ramanathan and Andersen [RR93], [And93], etc. They are discussed in section 4, Review of Literature and Prior Related Work. In the articles reviewed and listed in the Bibliography, Site Synchronization has been studied from the perspective of individual sites with access to sufficiently large buffers of data to permit investigators to assume that they can start at some reference point in the stream and ‘play’ as many frames of information as required. In a multimedia environment where there are requirements for large quantities of data and many streams, buffering alone may not be sufficient to provide the Inter-stream Synchronization capabilities required.

The issues involved in the coordination of multiple synchronous datastreams from multiple origins to multiple recipients that are addressed in this thesis are:

1. The Ensemble expression must allow a means for identifying/specifying the streams; M, N, O... and the number of data packets (x, y, z,...) of each stream that is part of the Ensemble.
2. The Ensemble expression must allow for starting streams at various offsets from each other.

3. Data-paths in the conference must be changeable.
4. Performance (delay times) must be measurable.
5. Feedback from Intermediate Synchronization Servers (ISS's) must be recognized.
6. Since ISS's are not dedicated to an application, they must be unaware of the content of the data.

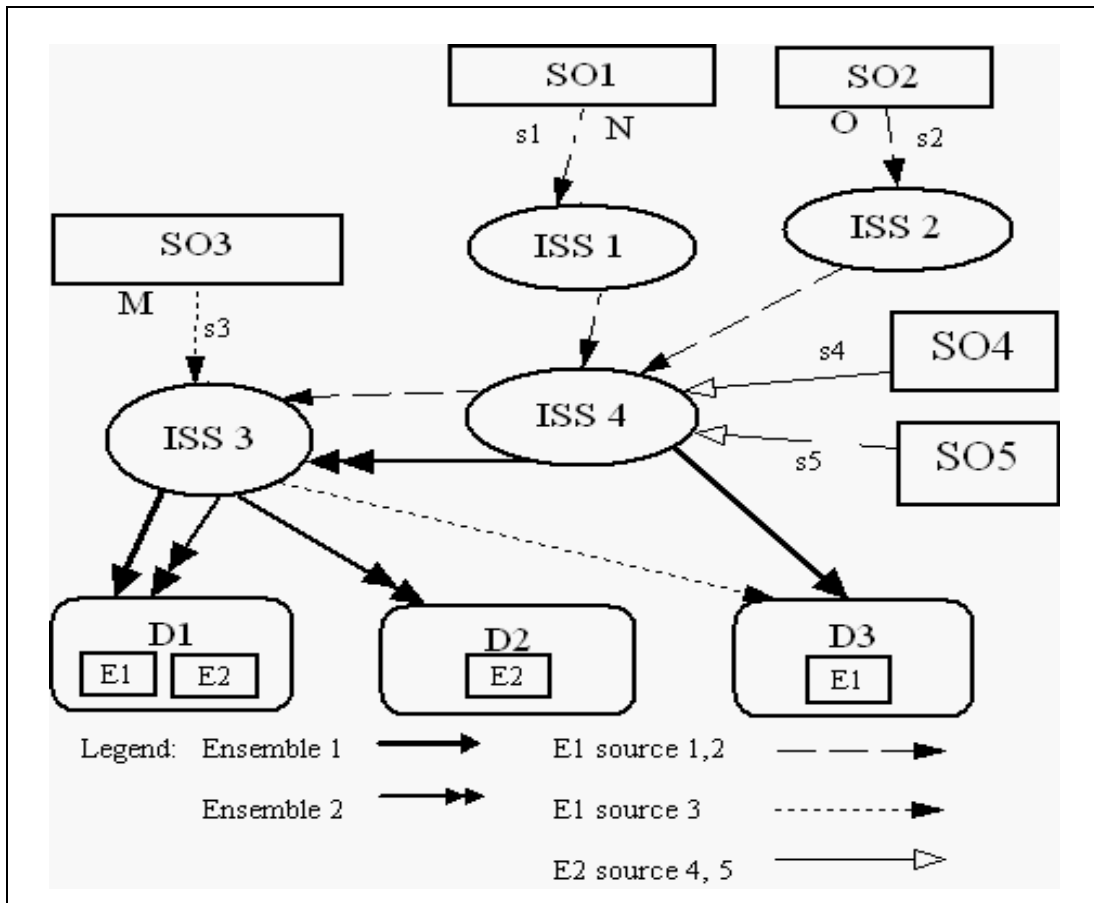


Figure 6. A multi-origin, multi-destination conference

Since the ECP is unaware of the content of the transmitted data, it is the responsibility of the application using the ECP, to either make transmitted record divisions sufficiently small as to automatically incorporate intra-stream synchronization or to include time-related data for the

stream within each record and manage the synchronization itself (which all current applications already do). The ECP then maintains the required Inter- and Intra- stream transmission, such that the records are presented to the application with the required coordination.

### **3.5. Objectives**

The overall objective of this thesis is to implement an architecture that supports the coordinated transmission of the elements of multiple synchronous datastreams (Ensembles) of a multimedia conference, to reduce delays experienced in the transmission of these Ensembles, while maintaining their inter-stream ordering.

In the remainder of this thesis, we describe the ECP program which was designed to meet the following objectives:

1. To determine the arguments and form of the expression,  $E_{M, N, \dots}(x, y, \dots)$  so the program can collect the elements of the Ensemble at multiple locations.
2. To provide an ability to dynamically specify Ensemble Coordination parameters using the expression  $E_{M, N, \dots}(x, y, \dots)$ .

3. To describe an algorithm for Ensemble Coordination that can provide the 3 Ordering Properties across multiple collections of streams of data, such that the objectives of Type I (Intra-object), Type II (Inter-object) and Type III (Site) Synchronizations and a capability for Type IV (Global Synchronization) can be provided by an application.
4. To demonstrate that Ensemble Coordination can be accomplished across multiple sites.
5. To provide Ensemble coordination in a programming layer that does not require adjustments from the presentation layer of the application or modifications to the underlying operating system or transport mechanism.
6. To include a capability for the application to dynamically specify the values of the expression,  $E_{M,N,\dots}(x, y, \dots)$ , as discussed above and hence, to control the ECP.

The gist of the idea is to provide an interface for an application to specify the elements of an Ensemble, and their coordination parameters, with the actual control being accomplished by a program that operates independently of the application. The ECP is the program that accomplishes these objectives.

### **3.6. Approach to the Solution**

The implementation of the ECP is based on my algorithm (described in more detail later in section 7 on page 81) for the processing of Ensembles. The basics of the algorithm are:

1. Determine the routing for each stream from the graph of the network.
2. Determine when all segments of the Ensemble are available.
3. If the current node is a Source or ISS node:
  - a) Determine the list of immediately adjacent nodes that are in the routing list.
  - b) Transmit the Ensemble.
4. If this is a Destination node, display the data.
5. Compute the new destination as in 3 above and transmit the data.

The basic mechanism for computing the routing (described in detail in section 5.10.6 on page 72), is to:

1. Compute the immediately adjacent nodes.
2. Perform the logical AND of the segment header, which contains the path list for the segment, with the adjacency list, then
3. Send the segments to the computed nodes.

Determining when an Ensemble is ready for transmission or display is controlled by a segment counter for each stream in each Ensemble.

Since an Ensemble transmission requires that a specific number of segments of each stream be transmitted together, before the next Ensemble can be transmitted, I had to be able to specify the number of segments of each stream of an Ensemble. In order to describe this relationship more clearly and provide a foundation for the implementation, I developed the following expressions:

$$C = \sum_{e=1}^m E_e$$

and

$$E = [(S_n, Q_n), \dots]$$

where  $C$  represents the conference,  $E_e$  represents the  $e$ -th Ensemble,  $S_n$  represents the  $n$ -th Stream in  $E_e$  and  $Q_n$  represents the quantity of segments of  $S_n$  that are required in  $E_e$ .

These expressions clarify the relationship between the segments of the streams of each Ensemble and are the basis for the implementation to specify the elements of an Ensemble at execution time.

Figure 6 on page 27 is the conference network use to test the ECP. This network arrangement was designed specifically to apply stress to the coordination mechanisms of the ECP. In this figure, we see an example of three data streams,  $M$ ,  $N$  and  $O$ , originating at separate locations,  $SO1$ ,  $SO2$  and  $SO3$ . For the definition of the conference under investigation, these three streams are required to be presented in a synchronized fashion at 2 final destinations. This forms an Ensemble Set. I defined a presentation ordering dependency among the streams,  $E_{M, N, O}(x, y, z)$ , where  $x$ ,  $y$ , and  $z$  represent the number of data packets of each stream that must be displayed as an Ensemble. The ordering dependency of the streams is inherent in the relationships of the streams to each other in the conference. The presentation at destination  $D1$  may or may not be simultaneous (in a ‘universal time’ sense) with the presentation at destination  $D3$ , although this might be an ideal result.

In order for the data of this conference to be displayed so that it looks like the original data, the Ordering Properties previously described, must be honored. Ordering Property 1 is needed to ensure that all elements of stream 1 arrive at D1 in the proper sequence to maintain Ensemble E1. Ordering Property 2 is needed to ensure that the required elements of streams 1, 2 and 3 arrive together at D1 for the conference application to maintain the Synchronization of Ensemble E1. Ordering Property 3 is needed because D1 and D3 are both destinations for Ensemble E1.

There are 2 additional data streams (s4 and s5) and their sources (SO4 and SO5) in the implementation test. These streams, forming Ensemble 2, were added to determine if the insertion of their segments would interfere with the segments of Ensemble 1 and to ensure that multiple Ensemble Sets could be managed simultaneously.

The ECP implementation provides the desired coordination covering the following areas:

1. A means to specify the elements of the Ensemble such that the Ensemble can be formed even if the elements originate at physically different locations.
2. A mechanism to dynamically specify Inter-stream coordination parameters. This was accomplished by the expression  $E_{M, N, \dots}(x, y, \dots)$ , in light of the work by Ali [AWG94], and Haindl [Hai96]. This information was encapsulated into an Ensemble transmission header packet.

3. A means to transmit the Ensemble to multiple recipients, with special regard to the work by Duato, Lin, McKinley and Robinson [Dua95], [LMN94], [MX\*94] and [RMC95]. The work of these authors may lead to a better design of the Ensemble header above.
4. A means to merge elements (see Figure 6 on page 27) from multiple sites, say, Stream Origins SO1, SO2 and SO3, into an Ensemble transmission proceeding from a third site, ISS3, to several possible destinations, such as Destinations D1 and D3.
5. A set of rules to determine how each ISS (Independent Synchronization Server) determines which streams to send and in what directions to send them. For example, in the arrangement of nodes in Figure 6, I wanted ISS2 to forward stream O to ISS4 instead of sending it directly to its intended receiver, Destination D2. This was done to test the segment gathering ability of the program and to make sure that a node could detect that it was not responsible for some of the streams in the Ensemble Set. It is important to note that this implementation assumes the existence of the graph defining the transmission paths. That is, there is no attempt to determine the graph purely from network node information. The graph's nodes and edges are supplied as information in a file and the LEDA Graph object is constructed from this data.
6. A means to quantify the effects of the expression  $E_{M, N, \dots}(x, y, \dots)$  on the delay of data at receiving nodes.

## **4. Review of Literature and Prior Related Work**

In the Lakes architecture [ABLM95], a mechanism is provided for achieving consistent interleaving of data from multiple sources at multiple receivers. The Lakes architecture does not deal with providing consistent inter-object synchronization at multiple sites. There is a discussion of a Service Request Manager, and a Process Planner working in a Server Group. The Process Planner executes 'scripts' which act like requests for service. Discussion of multicasting is only in relation to the underlying support in Amoeba and V (operating system level multicast) and Isis (application level multicast).

Akyildiz [AY96] defines a protocol for synchronization of sites that are defined to be in a group. Definition of a group is not given. Definitions are given for intra- and inter- media synchronization. Intramedia synchronization is restricted to synchronization within a single medium. Intermedia synchronization is synchronization (a) across multiple connections, or (b) interleaved media on one connection. No work is mentioned on intra-media synchronization. Intermedia synchronization is restricted to one connection. In the approach taken by Akyildiz, a negotiation phase among all members of a group determines a global time reference point for the group called Virtual Global Time (VGT). Media units are collected at a site(s) and distributed to the group. A media unit is presented at all nodes in the group at the same global time. Packet coordination is controlled by time-stamps and delay-stamps. This approach is different from the approach taken in this

dissertation. In the approach taken in this dissertation, there is no concept of global time and there is no central server.

Ali [AWG94] develops a Synchronized Virtual Channel Architecture (SVCA), Quality of Presentation (QOP) parameters, equal sized Atomic Units of Presentation (AUP), and uses Intermediate Synchronization Servers (ISS), with the goal of preventing skew at destination sites. Omitted is the strict performance criteria required in 'live' multimedia (as in video-conferencing). Resource requirements are scheduled at connection time. Data flows on the established SVC only, and synchronization is only performed at ISS's. Buffering requirements are discussed and formulae are developed to determine minimum values that will prevent over-run. Jitter is reduced by buffering at each ISS, which "gives an illusion of a constant delay channel that satisfies the desired QOP."

Anderson [AH91] discusses an implementation of logical time systems, with rate adjustment of streams. Processes and objects are presented in the context of the ACME system. Anderson later defines a Continuous Media (CM) Resource model[And93] that is the basis for a meta-scheduler where resources, required to present CM data at a destination, are reserved in advance. The model defines parameters that must be satisfied to eliminate device starvation. This includes formulas for calculating buffering requirements for the data. Although this work does mention an extension for multicasting the data from a single source to multiple destinations, it does not deal with synchronization of multiple destinations. Anderson defines "integrated CM" as a transmission in digital form, using the same hardware and software as other data in the system. Scheduling policies are managed via use of a "meta-scheduler" which makes "reservations"

within the Operating System for a guarantee of service rates, although there is no discussion of the coordination of transmission across multiple streams, nor is there discussion of how the operating system is to implement the guarantee. There is a discussion of elimination of starvation due to jitter. A “Workahead” concept is implemented via buffering. There is no discussion of the problems associated with multicasting.

Ballardie [BFC93] presents a new scalable multicast architecture that is de-coupled from, but dependent upon, unicast routing. The key concept in Ballardie's paper is the design of Core Based Trees, wherein packets are only replicated when paths diverge from a router. The concept of limited routing is also used in my thesis, although the implementations are quite different.

The Tenet Real-Time Protocol Suite discussed by Bannerjea in [BKTZ94] addresses the stringent performance requirements of multimedia applications. The suite uses resource management, connection control and packet control to achieve this goal. The Tenet paper does not address Inter-stream Coordination or Synchronization.

As previously mentioned, in (section 2.3), Blakowski [BS96] defines 2 types of synchronization, including inter- and intra-object synchronization and “lip-sync,” in multimedia systems. Blakowski discusses “live” vs “synthetic” synchronization and multi-stream synchronization with reference to IBM's MMPM/2 (MultiMedia Presentation Manager) support. Multicasting is relegated to the responsibility of the stream layer and there is no discussion of multiple sources.

Courtia [CFC\*96] presents a formal approach for specifying causal relations associated with a stream's information units that express the delivery constraints of that information unit relative to the delivery of other information units that may belong to the same stream or other streams. This work addresses only the situation where there is a single destination, so multi-casting is not addressed.

In the papers by Duato, Lin, McKinley and Robinson [Dua95], [LMN94], [MX\*94], and [RMC95], the authors explore the possibilities of wormhole routing in multiprocessors, wherein each message contains a header consisting of a string of bits used to control the flow of information in mesh-connected multi-computers. These *'flits'* (flow control bits) help to avoid buffering of messages as they progress through the network. This concept was instrumental in the decision of how to implement the header information for the ECP.

Garcia-Molina [GS91] concentrates on the ordering of messages but does not specify the relationship of delays in multiple streams. I have relied heavily on the definitions provided in his paper and have discussed them extensively in section 2.2 on page 7.

Haindl [Hai96] presents a hierarchical synchronization model for describing time relations necessary for the presentation of time-dependent data. The concepts relevant to this paper are: an inter-stream synchronization model for predicting presentation time of a "data unit", the mathematical specification of intervals, and a notation for an ordered set of subsets - the synchronized data units of multiple data-streams. The issue of multiple destinations for the time-dependent data is not addressed, nor is there any discussion of general multicasting problems.

Ishii's work [IM91] describes the evolution of a Computer Supported Cooperative Work (CSCW) system. It began as TeamWorkStation (TWS) and evolved into ClearBoard (CB). There were two versions of TWS and two versions of CB. In this system, the target of collaboration is a desktop. The image of the action occurring on a desktop is combined with the image of the participant sitting at the desk and the two images are transmitted as a single data stream. The object of collaboration and the audio and video of the participant make up a single object. It was implied that the system only supports two participants. It does not deal with synchronization of multiple sites.

The work done by Jardetzky [JSRM95] is similar to Anderson's work [And93]. It deals with intra-stream synchronization. It defines Intra-stream Synchronization as the correspondence between media units and the passage of time. Like Anderson's work, it is concerned with the elimination of jitter. Mechanisms for Inter-stream Synchronization and event synchronization are also described. This work deals with a single source and a single destination. This paper does not deal with synchronization of media presentation at multiple sites. Jardetzky proposes "stream agents" which perform inter/intra-stream synchronization, with support of "group agents" in a "locally distributed area." There is no discussion of multi-casting.

Kleinholtz [KO94] describes a multimedia conferencing application called Bermed, under construction in Berlin at the German Heart Institute and the Rudolf Virchow University Hospital. Collaborative objects for the conference are all time independent. Pointing to, and manipulation of, the objects are supported. The audio and video streams of the conference participants are the only time-dependent data supported by Bermed. Kleinholtz does recognize the need for synchronization

of the application input/output and telepointing events, but it is

Rothermel, in [RH96], provides a mechanism for specifying the synchronization of multiple media using a “clock abstraction” which relates media time to real time. There is no provision for Site Coordination or Synchronization, nor is there a mechanism for coordinating transport layer activity.

Schnepf [Sch94] describes a model for specifying coarse-grain (begin/end) synchronization between multimedia objects. This is a base for multimedia authoring tools. The model does not deal with multiple sites.

Vin [VZSR91], in describing the Etherphone system, discusses the delays for end-to-end propagation and processing, for audio only. There is no discussion of video, multiple streams or multicasting.

The paper by Yu [YWS95] deals with pause/resume support in a Video on the Demand (VOD) system where multiple users share a single stream. There is no Coordination or Synchronization of users. Users can independently view the stream of data and can pause/resume independently. If a user pauses, at the time of resume, the user is no longer viewing data from the original stream. Depending on the length of the pause, either the data comes from a buffer that was stored from the original stream, or another instance of the stream is initiated in the place of a recently completed stream.

## 5. System Architecture

### 5.1. Notation Conventions

In the following sections, *program names* are in bold, italic typeface; *data structures* and *file names* are in italics, and **functions** and **methods** are in bold. C library, C++ library, LEDA library, Pthread function calls and Operating System calls are embossed. Simple variables are in Arial typeface.

### 5.2. Network Conventions

The ECP operation relies on knowledge of the network connecting its source and destinations. Specifically it must be able to process the graph of that network. The ECP does not determine the graph itself, but relies on external input to describe the nodes and edges of the graph.

Within the graph of the network, nodes are defined as having 4 basic types: Source, Destination, Independent Synchronization Server (ISS) and a combination of Source and Destination. An ISS provides a means of merging and forwarding portions of an Ensemble to provide the routing capability for Ensemble data in a way that emulates multi-casting, reduces buffering and eliminates re-broadcasting to nodes that do not need the data.

For purposes of simplification, an ISS is not allowed to be either a Source or Destination, and thus provides only Ensemble coordination and network store and forward activities. Since a physical node can participate in a conference as both a Source and as a Destination, there is a need to distinguish the roles being performed at any given time. To accomplish this, we refer to each role as a *virtual node*, and build the network graph based on the virtual nodes of the conference.

### **5.3. Restrictions**

Due to the nature of this implementation, its purpose and the need to reduce the implementation effort, the following restrictions have been imposed:

1. The conference configuration (the graph) for the current implementation is static.
2. Defined nodes always participate for the entire test.
3. Nodes cannot be dynamically added or deleted.
4. Ensemble rules are currently static, from conference start to conference end.
5. Ensemble formation rules are static from conference start to conference end.
6. Network partitioning is not considered a problem to be managed here.
7. Nodes may not currently change IP address during the conference.
8. Conference data streams and their sources are determined before each node's program is started and cannot be modified during the conference.

The design of the program allows straightforward extensions to remove all the above restrictions.

## **5.4. Security**

Two files specify the location of participants: *ne.dat* and *node2ip.dat*, described in detail in section 6 on pages 75 and 76. In the current implementation, only nodes that are listed in the *ne.dat* file can participate in the conference. These nodes are listed in the *node2ip.dat* file with their IP addresses. Each node has a copy of both *ne.dat* and *node2ip.dat*. The *node2ip.dat* file provides a mapping from domain names to IP addresses for systems that are not supported by a nameserver. The program listens on port 49999, which is currently unassigned to a specific well-known application. This value is specified in *ne.dat*, so it can be easily varied before the program is initiated. The *ne.dat* and *node2ip.dat* files must be propagated to all conference sites before the conference is initiated.

## **5.5. Multi-platform Considerations**

This program was designed to run on multiple operating systems and multiple hardware platforms. The decision to do this provided several challenges, including: selection of multi-tasking mechanisms and control thereof, programming languages, network interfaces, and error-handling code. These challenges are discussed in the following sections.

### **5.5.1. Multi-tasking**

After reviewing the differences in programming processes vs. threads and the overhead involved in context switching for processes, as well as the lack of standardized cross-system interfaces for inter-process communication, I decided to use threads. Threads do not use context switching, but do allow for global read/write data. This removes the overhead of inter-process

communications. Also, as discussed below, there are standardized cross-platform mechanisms for signaling and providing for mutual exclusion with threads.

This left the problem of which kind of thread mechanism to use. Microsoft<sup>®</sup> provides its own thread control mechanisms (and function calls), which are not source-code compatible with the 2 mechanisms available in the UNIX environment (POSIX<sup>®</sup> threads and Solaris<sup>®</sup> threads). All three mechanisms have their own runtime libraries and different structures for the function calls that invoke the creation and other manipulations of threads.

In the Microsoft (MS) Windows environment, there are two readily available thread implementations: the native MS Windows thread management mechanism, and the Pthreads for Windows package described in [But97] and [Joh]. The interfaces of Pthreads are defined in [But97] and [LB98].

The MS Windows implementation exhibits some minor differences between Windows NT<sup>™</sup> and Windows 98<sup>™</sup>, mostly in certain security options, but the major downside of the Microsoft thread management interface is that it is not source compatible with either of the UNIX mechanisms. In fact, there are major differences in call sequences (such as: how to create a thread, the arguments required, and even the data types of the arguments). The only advantage of the MS implementation of threads is that it *appears* to be similar to the implementation of threads in OS/2<sup>©</sup>. OS/2 is mentioned because, at the time when this research was first contemplated, OS/2 was to have been a candidate as a platform to support the program. Since the decline of OS/2 as a widely accepted and well-supported operating system, the motivation for this extension has

been removed. OS/2 would have also required the extension of LEDA to that platform, an effort that had been underway in Germany (where LEDA was developed), but was dropped for the same reason of usefulness as well as the cost to the LEDA development team. The extension of LEDA to the OS/2 platform may be continued as an independent extension to this project in the future.

In the UNIX environment, there are two different thread packages: Solaris threads and POSIX threads. The tradeoffs in the two different thread packages as presented in [SUN] are as follows:

- POSIX threads are more portable.
- POSIX threads establish characteristics for each thread according to configurable attribute objects.
- POSIX threads implement thread cancellation.
- POSIX threads enforce scheduling algorithms.
- POSIX threads allow for clean-up handlers for fork calls.

versus,

- Solaris threads can be suspended and continued.
- Solaris threads implement an optimized mutex and interprocess robust mutex locks.
- Solaris threads implement daemon threads, for whose demise the process does not wait.

Since threads can be "suspended" using various "wait" calls, optimization of mutexes is not critical to the ECP, and because the ECP does not use daemons, Solaris threads do not appear to provide any capabilities that were not available in POSIX threads. On the contrary, POSIX threads provide a significant number of additional capabilities including:

- Thread attribute management,
- Thread canceling and cleanup,
- Mutex attribute management and
- Semaphore 'open', 'close' and 'getvalue'.

While it was not clear at the initial design of this program that any of these additional capabilities would actually be required, their presence was certainly a factor in making the decision as to which type of multi-tasking would be used and which thread package was preferred.

With these considerations understood, the decision was made to use the Pthreads and POSIX implementations. This meant that, once written, the code would operate using identical source and semantics in all of the selected Operating System environments: Windows 98/Me/NT, SunOS 5.6 and above and in possible future implementations, such as Linux. As previously mentioned, I also thought about including OS/2 as a platform but the viability of that Operating System is in serious doubt and its use has severely declined since this work began. Nevertheless, there are a few of the MS Windows thread calls left in the source code as comments for future reference, since they are similar to the OS/2 thread calls.

The pthread calls used in the Ensemble program are found in the Pthreads for Windows (freeware) package[Joh] and are fully compatible in source-form with the POSIX threads implementation of those calls in UNIX [But97].

### 5.5.2. Programming Languages

When considering multi-platform implementations, the choice of a programming language is severely limited. There are no languages that provide a common windowing interface across operating system platforms. Some languages do not provide any such interface; most do not provide built-in threads or communications primitives (requiring that programmers resort to assembler language subroutines, cross-language interfaces, or external support packages). While Java provides all of the preceding capabilities, as well as object-oriented facilities, it does not provide access to a readily available template library or library package that provides manipulation functions for graphs. There is such a package for C++ called LEDA, the Library of Efficient Datastructures and Algorithms [LEDA1]. The LEDA library provides a complete set of classes and methods compatible with, and extending, the Standard Template Library. LEDA includes a set of built-in algorithms for manipulating these data structures, a set of methods for working with graphs, dictionaries, maps, sets, and other more arcane objects, and most importantly: a system independent mechanism for drawing and writing textual and graphical windows, using either MS Windows or X-windows as the underlying implementation [LEDA1]. An additional advantage of the LEDA implementation of windowing support is that the user does not have to think about the underlying mechanism, insofar as getting data in and out of the system. While the human interfaces of X and MS Windows are different, as well as there being a difference in the client-server model (the two methods are reversed in concept with each other), the basic user mechanisms are consistently presented and the differences are not apparent to the user. The programmer is also protected from these differences and has a single consistent Application Programming Interface to all windowing operations.

Thus, with the selection of Pthreads and LEDA to provide the tasking, data structure, and windowing control capabilities needed, it became possible to implement a program that would operate on multiple platforms with no changes to the code and extremely few compile-time variations. These variations are easily handled with the C++ pre-processor mechanisms: *#include* and *#ifdef* for inserting the proper error-handling and networking library modules.

### 5.5.3. Network Interfaces

The network interfaces in UNIX and Windows are slightly different at the application level, primarily in the options required for the various system-level function calls and in error-handling. The programming differences are handled as one- or two-line code differences, handled with the *#include* mechanism of C++.

### 5.5.4. Error Handling

The error handling differences are twofold: in UNIX, error settings are placed in a global variable called `errno`, the value of which must be recorded before any additional system calls are made. In Windows, the last error is available as either a value set in a call parameter in the called function parameter list, or from the Windows function call named `GetLastError`. The other difference in error-handling is that the names of the error codes (used to avoid hard-coded error numbers) are different; for example, in UNIX one might reference the TCP/IP error code `EINPROGRESS` and in Windows it would be called `WSAEINPROGRESS`. This requires the ECP to have two lists of error codes for detecting problem causes.

## **5.6. *Distributed Execution***

Since this program is designed as a set of peer-to-peer programs, a mechanism was needed to initiate the program at each node of the network, pass it runtime values and display its actions, all from a single computer. Two facilities have made this possible: Exceed™ for Windows, from Hummingbird® Software, and the (freeware) Virtual Network Computing (VNC)© facility, by James Weatherall at AT&T Research Labs, Cambridge.

Exceed provides the ability to display multiple UNIX Xwindows and Xterms on an MS Windows computer, while VNC provides the ability to simulate the entire desktop of a remote Windows NT system in a single MS Windows window. VNC is started for each of several NT systems, each running in its own window.

## **5.7. *Basic Program Description***

All computers participating in the conference operate in a traditional peer to peer relationship. Each computer acts as both a client and a server. Producer-Consumer code was used in several places to ensure operation overlap and proper buffer handling.

The program is composed of 2 layers, the operational layer and the application layer. The operational layer is arranged in 2 stages: preparation and execution.

The first stage is concerned primarily with establishing a network connection, creating the internal representation of the network graph and then obtaining the rules for the Ensembles.

The second stage of the system performs the actual coordination of the datastreams, building the Ensembles according to their rules, transmitting and receiving them, and making them available to the application layer.

The Ensemble rules are the values that specify which streams are included in which Ensembles and how many data segments of each stream are to be transmitted (coordinated) with each other to form those Ensembles. The details of specifying these rules is discussed in section 6.2 on page 76.

Stage 2, the execution stage of the program consists of the receiving and transmitting threads and the input/output simulations.

The receiving threads are concerned with managing the incoming segments of the Ensembles that arrive at the machine and dispatching them to either the conference application or the transmitting thread for output to the next destination for the Ensemble. The Ensemble header in each incoming data segment contains routing information. If the information indicates that this segment is to be displayed at this machine and this machine is a destination machine, the data is queued for output in an Ensemble window. If the current machine is an ISS, the data is enqueued for output to its next destination(s).

In the transmitting thread, the data header in each record is inserted (for new records) or is utilized on an ISS (from incoming records) to determine the next destination for the record. As described in the algorithm in section 7 on page 81, the computation is inherently simple. During Stage 1, the Pathnode Vector was computed for each stream and saved for use in Stage 2. During Stage 2, when all records for an Ensemble become available, the program examines the header of each record of the Ensemble. If the current record *does not* originate at this machine and the current machine *is not* a final destination for this record, the program will:

- a. Extract the Pathnode Vector from the record header
- b. Perform the logical OR of the extracted Pathnode Vector with the Distribution Vector saved from Stage 1
- c. Send the record to those nodes corresponding to the '1's in the result.

If the current record *does* originate at this machine, the Pathnode Vector computed during Stage 1 is attached to the header of the record and the previous 3 steps are performed.

If the current machine *is* the final destination for the record, the record is sent to the conference application.

In order to simplify the implementation, the actual conference application is simulated by the file input program (*Q\_E\_Files*) and the window display programs (*win\_dispatcher* and *draw\_window*).

## **5.8. Implementation Stage 1**

Stage 1 consists of the following major sub-programs, in order of execution: *Build\_graph*, *create\_graph*, *Build\_E*, *Start\_E\_window\_mgr*, *E\_window\_mgr*, *Start\_receiver*, *Start\_select\_thread*, *Start\_sndthread*. These programs create and/or initialize the content of the data structures discussed in detail in section 5.10 on page 69. All conference nodes must complete Stage 1 before the conference may proceed. When all the nodes are ready, a signal is sent from a central node to inform all nodes that Stage 2 may commence. This is manually initiated and is the only centralized action in the conference. We now describe the details of the actions in Stage 1.

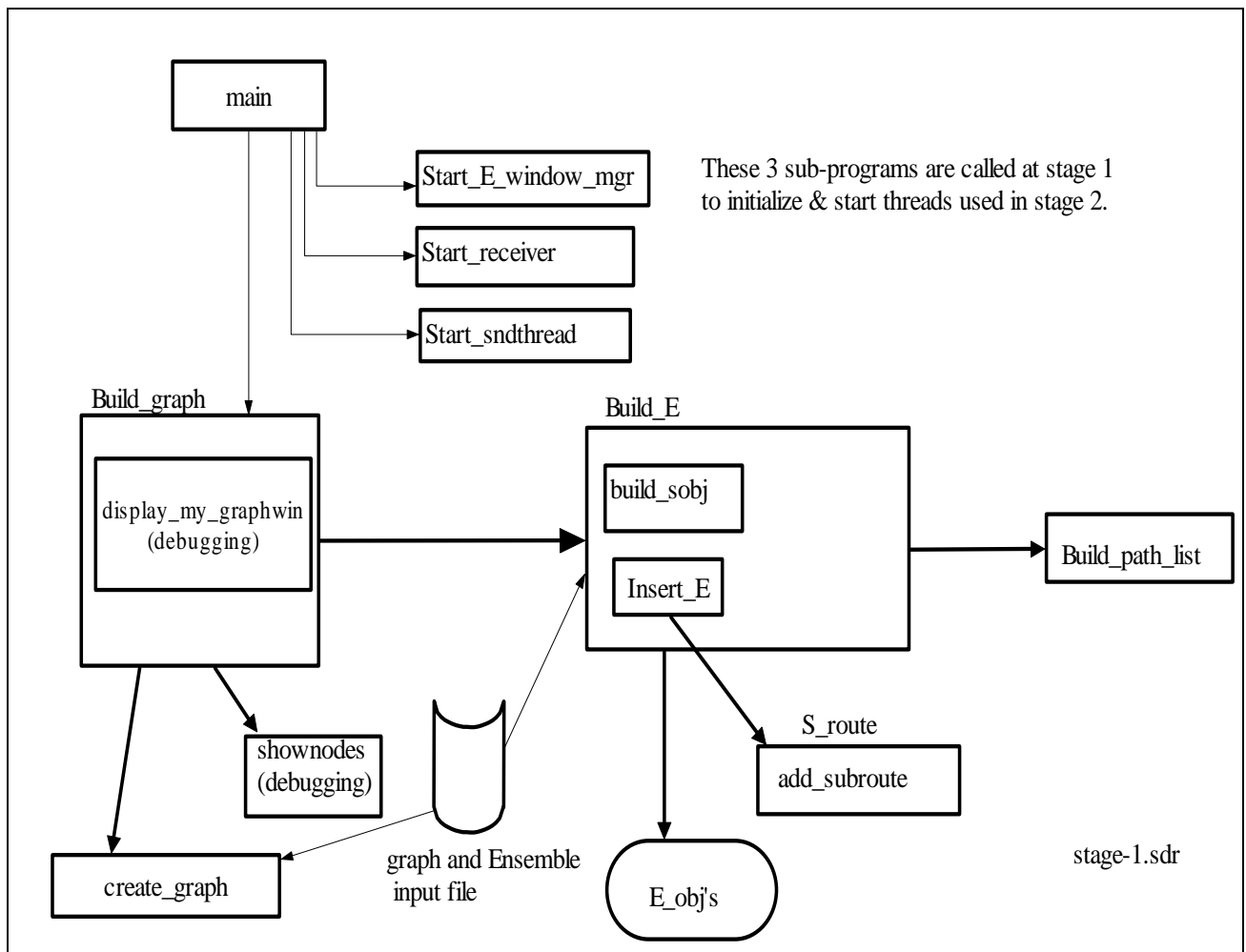


Figure 7. Program Architecture, Stage 1

### 5.8.1. Initialization

The initialization of the various structures and objects is a complex process, involving constructing a graph of the conference network, determining the rules for Ensemble coordination, providing communication with the application layer, and performing the common TCPIP network initializations. The following sub-sections describe the programs that perform these initialization activities.

#### 5.8.1.1. *Build\_graph*

Stage 1 starts with determining the graph structure and inserting the Ensemble Objects into the nodes. ***Build\_graph*** is responsible for all operations concerning building the conference's network graph and the routines that initialize the data structures within each node. ***Build\_graph*** first calls ***create\_graph***, which reads the file (*ne.dat*, see section 6.1) that describes the nodes and their directed edges. (Dynamic graph construction is beyond the scope of this work, but it is noted that in Bracken's thesis [Bra93], a conference invitation protocol was defined which, together with existing network search programs, could be used for that purpose.) ***create\_graph*** builds the digraph structure as a LEDA "graph" class object named *Gnodedata* (Figure 9. The Graphnode & E\_obj Descriptor Structures). After the graph has been constructed, ***Build\_graph*** calls ***Build\_E*** to create and initialize the Ensemble objects that belong to each node.

#### 5.8.1.2. *Build\_E*

***Build\_E*** reads the list of Ensemble names and the data streams which they contain, as well as the definitions of the source and destination nodes of each stream from the *ne.dat* file. Each Ensemble and Stream is assigned a number when this file is processed. This information is incorporated into the *E\_obj* and *S\_obj* structures (see details in Figure 9. The Graphnode & E\_obj Descriptor Structures). Since these operations are performed for every virtual node, every node has all information about every other node, every stream and every Ensemble, thus supporting a future modification for complete dynamic re-configuration. ***Build\_E*** also inserts data in a 2-dimensional array called *Required*, which specifies how many segments of each stream are required for each Ensemble. The indices of this array are the Ensemble number (the row index) and the Stream number (the column index). If an Ensemble is NOT relevant at the

current virtual node, all columns of *Required* for that Ensemble are set to zero. This prevents the program from waiting for streams that will never arrive.

When *Build\_E* has read the real and virtual names of all the source and destination virtual nodes, it does a graph traversal for each pair (source/destination) and builds a list of all the virtual nodes found on each path from the source to the destination. The traversal is not necessarily minimal. It is only required to follow the digraph edges. These lists are stored in the *E\_obj* as an *S\_route*. There is one *S\_route* for each stream per Ensemble. An *S\_route* contains one *Pathobj* for each different path that a stream can take. For example, given the digraph A->B->C and A->E->F, stream s1 could go from node A to B to C as well as A to E to F) for each Ensemble, so C and F would both be displaying the Ensemble containing stream s1. When the *E\_obj* has been completely initialized, it is inserted into the graph structure at the node being initialized.

At this time, two additional 2-dimensional arrays are constructed. These are *Required\_S* and *Avail*. These 2 arrays, along with *Required*, are used to determine when a sufficient number of segments have been enqueued (the actual number available) for a given Ensemble/Stream combination so a signal can be issued to the output queue manager (*Out\_Qmgr*) or window dispatcher (*win\_dispatcher*). *Required\_S* is used for initial transmissions of stream data from a Source node. *Required* is used for re-transmit or output requirements. *Avail* contains the count of all currently available (enqueued) segments of all streams in all Ensembles at the current node.

The object class, *Gnodedata*, maintains all the information about every node, and includes multiple occurrences of the class Ensemble, one for each group of datastreams to be processed

by the node. A *Gnodedata* object is accessed from the parametric data at a node, *n*, of the Graph, *G*.

#### 5.8.1.3. *Start\_E\_window\_mgr*

*Start\_E\_window\_mgr* is responsible for starting (possibly multiple) application output simulation setup thread(s), *E\_Window\_Mgr*. During Stage 2, the threads receive control asynchronously to insert data into their windows. *Start\_E\_window\_mgr* examines the *Ensemble\_list* in the *Gnodedata* for the current node. For each Ensemble found, an *E\_Window\_Mgr* thread is created and started. In the situation where there is a real application, *Start\_E\_window\_mgr* would call the application to get the addresses of the functions to receive control from upon receipt of an Ensemble.

#### 5.8.1.4. *E\_window\_mgr*

Each *E\_window\_mgr* runs as a thread and displays all the data for one Ensemble. An *E\_window\_mgr* thread is only created for an Ensemble that is to be presented as output at this node. Each *E\_Window\_Mgr* thread creates a display window (*E\_window*) for an Ensemble for which the current node is a final destination. Since each Ensemble window is handled by a separate program thread, parallel display across multiple Ensembles is possible. If there were a real user application instead of a simulation, *E\_Window\_Mgr* would be replaced by a signal to the application at its segment handling interface for that Ensemble.

#### 5.8.1.5. *Start\_Receiver*

When all of the *E\_Window\_mgr* threads have been created, the main procedure calls *Start\_Receiver*. This program has 2 primary functions: initializing all telecommunications for incoming data and starting the program threads that manage the incoming data. *Start Receiver*

first calls *Create\_Bind\_Listen* does a `socket` call to create a TCPIP socket, performs a `TCPIP bind` to that socket on a specific port (port 49999, which at present, has not been officially assigned to a fixed owner), then begins the `TCPIP listen` operation, which prepares the socket for connection requests. *Start\_Receiver* now calls *Start\_select\_thread*, to perform some additional initialization for network communications and creates the thread, *select\_thread*. *Start\_Receiver* now creates two more threads: *Q\_E\_Net* and *win\_dispatcher*. The former performs the enqueueing of incoming data from all remote nodes. The latter determines which *E\_window\_mgr* thread is to display the data (when the data is destined for the current node). When *Q\_E\_Net* has enqueued a complete Ensemble, it sends a signal (via `pthread_signal_cond`) to either *win\_dispatcher* or *Send\_Ensembles* that data is ready. The determination as to which signal to send is governed by the type of node this is: if this node is an Independent Synchronization Server (ISS), then data must be forwarded to another node for display or further re-transmission. If this node is a destination (or both a Source and a Destination), then *win\_dispatcher* is called. If there were a real application, then *win\_dispatcher* would signal the application instead of an *E\_Window\_mgr*.

#### 5.8.1.6. *Select\_thread*

*Select\_thread* issues the `TCPIP select` request. The `select` request is in a loop, which waits for incoming connections. Since the loop is in a thread, the rest of the program may continue any other processing while the thread is blocked. When a remote location requests a `TCPIP connect` to the current node, *Select\_thread* is awakened and returns a descriptor for the incoming socket. *Select\_thread* then does a `TCPIP accept` and calls *Start\_receive\_thread* to create a thread (*rcvthread*) to handle the incoming data on a port assigned by the operating system. All further communication with that remote node will be through this same port, using this *rcvthread*. Thus,

each new incoming request from another node will have its own socket and its own thread to handle the communications. This eliminates a possible bottleneck in processing requests. The operating system will allow each thread a share of processor time. To enhance fairness (since the Operating System does not guarantee fairness), each *rcvthread* surrenders control via Pthread `sched_yield`, before doing another `receive`.

### 5.8.2. Preparation for Stage 2

During this stage, the ECP provides a differentiation among nodes. For the purposes of simplifying the implementation and control of the conference, one node is assigned the responsibility of ensuring that all nodes are on-line, and that the conference can proceed. (The Distributed Interactive Digital Multimedia program [Bra93], uses an "invite" facility, not implemented here.) This node is termed the "master" node. This obviates the need for writing code to manage the asynchronous startup of multiple nodes, which is not the subject of this investigation. When all nodes are available (their Stage 1 operations are completed), the operator of the master node starts the master node program, with a parameter that indicates it is the master.

When a node has reached the point where it has reached the TCPIP `listen` operation and is therefore ready to receive commands, it decides whether one or two command packets must be sent. If the node is running from a dynamic IP address, it sends a 'translate' command via the program `send_newaddr`. This command may be sent by any node before it completes Stage 1. It is used to inform other nodes that the sender is operating on a dynamic IP address and the current value of that address. The master node may send this address change message, then calls the

**send\_OK** function, which builds and sends a command packet to every node in the conference, including itself, to begin transmissions. This initiates Stage 2 on all nodes.

## **5.9. Implementation of Stage 2**

In this stage (see Figure 8 on page 61) the program performs all of its continuing operations: sending and receiving data, buffering segments of Ensembles, determining when Ensembles are either ready for re-transmit to another node or for output to the application on the current node and delivering the segments to these locations. The primary programs in this stage are: *select\_thread*, *rcvthread*, *In\_Qmgr*, *Q\_E\_Net*, *Q\_E\_Files*, *bfiller*, *Send\_Ensembles*, *Xmit\_data*, *Out\_Qmgr*, *sndthread*, *win\_dispatcher* and *draw\_window*.

Sending data is done by building a data structure defined by the *packet\_def* object. The *packet\_def* contains the data to be sent, control information describing the data and the Path Vector (see description on p. 73). To avoid problems with big-endian vs. little-endian systems and possible differences in the meaning of "integer" on different platforms all data in a *packet\_def* is marshaled. That is, lists are converted into sequences of strings and numbers are converted into strings of ASCII digit characters. Upon receipt, the data is un-marshaled. The first element of the *packet\_def* is a type-code, specifying what kind of packet it is. This allows for packets command and control packets, as well as data packets. The currently defined command codes represent "Begin Stage 2" and "Translate IP-address." The former is used to inform all nodes that initialization of the conference is complete. The latter is used by nodes with dynamic IP addresses or which are operating behind a router, to inform all other nodes of

their true current IP addresses. For data transmission packets, the packet contains the name of the stream this data belongs to, the list of Ensembles in which the stream is contained, the Pathnode Vector (*PV*) for the stream, a copy of the data and an end marker for the data.

In order to maintain a datagram-like capability in a TCPIP environment, the TCPIP NODELAY option is set during initialization of each socket. The expectation is that, according to TCPIP documentation, this will disable the TCPIP Nagle algorithm and cause each segment of data to be individually transmitted.<sup>3</sup> Unfortunately, due to the nature of some implementations of TCPIP on UNIX platforms, these data packages may not actually be constrained to individual segments of a stream. When acknowledgements of transmission to a remote site are not received in a timely manner these systems refuse to send the data and each successive `send` request causes the data to be appended to a system buffer. When the acknowledgement is received, this entire buffer is transmitted in a single operation.

Even when using the `TCP_NODELAY` flag to disable the Nagle algorithm (used to by many TCPIP implementations to combine `send` buffers), some UNIX systems will still buffer multiple packets, if network acknowledgements return too slowly. Current Windows systems appear to honor the `TCP_NODELAY` flag. Since the program is data-segment oriented and runs on Windows NT, Win9x and UNIX, it must be aware of the actual packet boundaries. An end

---

<sup>3</sup> The Nagle algorithm was added to TCPIP implementations to improve performance of the overall network, by reducing the intersite communication requirements between packages of data. This is completely in line with the concept that TCP is a stream-oriented communications medium, with guaranteed, in-order delivery. UDP provides datagrams, but does not guarantee delivery or ordering. Since reliable delivery was desired, TCP was chosen over UDP.

marker provides this boundary for the current implementation. The data length information is available in the record, but is currently not used to determine the boundary.

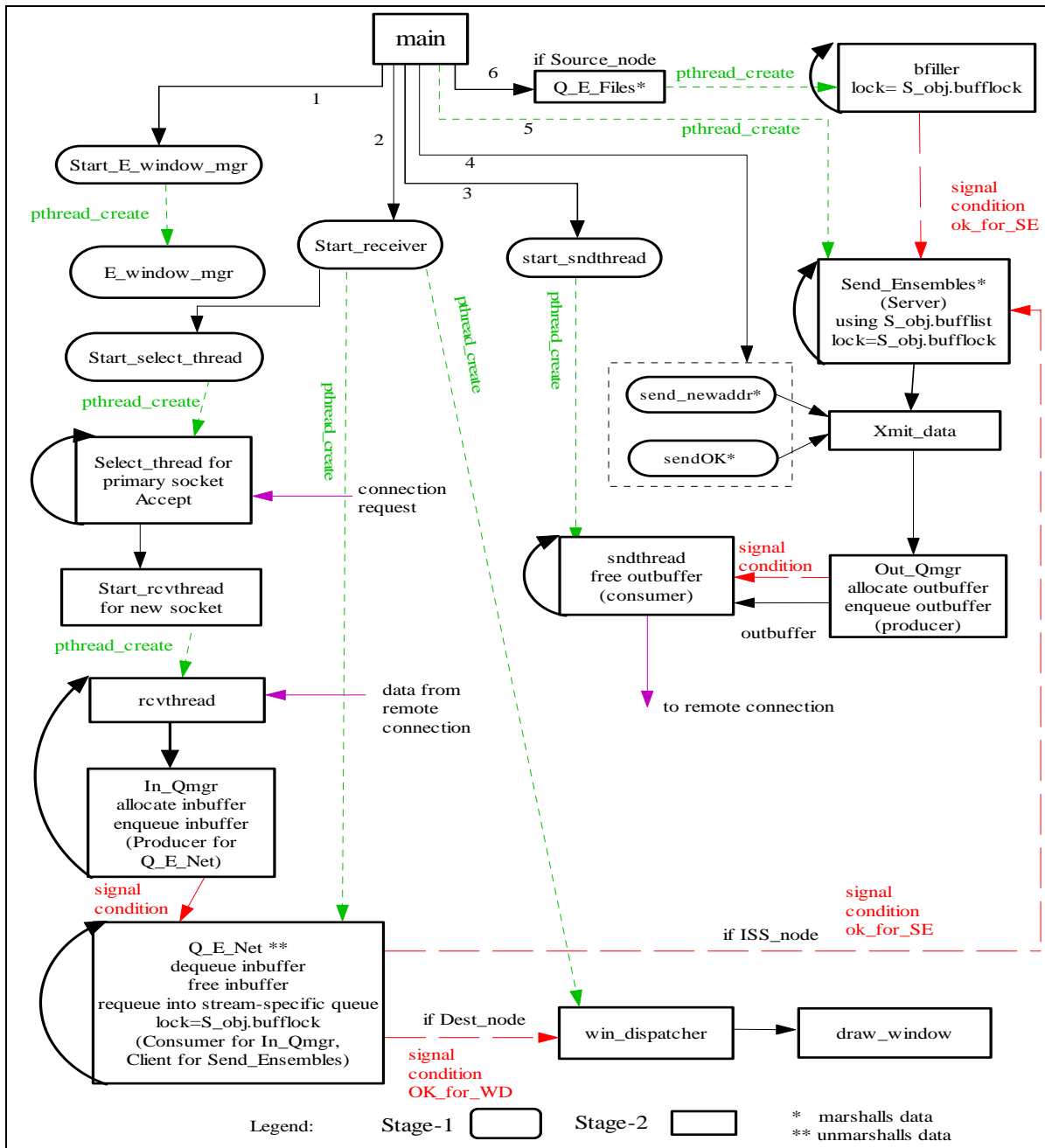


Figure 8. Program Architecture, Stage 2

### 5.9.1. Sending Data

If the current node is a source node for any stream(s), *Q\_E\_Files* is started to create threads to read the data for the streams. (Recall that the current implementation simulates a real conference with data from files.) These threads enqueue the data for output using the stream object (*S\_obj*) data class method, **add\_to\_bufflist**. If there were a real application, *Q\_E\_Files* would be replaced by a signal to the application telling it to begin enqueueing data for transmission. The **add\_to\_bufflist** and **get\_Eseg** methods in the *S\_obj* class, manage the buffers used for sending and receiving sequences of segments.

**add\_to\_bufflist** accepts a data segment and enqueues it in a list of segments for the Ensemble. The **get\_Eseg** function is called to retrieve one segment. When a segment is no longer needed by an Ensemble at this node, it is removed from the queue.

#### 5.9.1.1. *Q\_E\_Files*

Since the current implementation contains a simulation of an actual end-user conference application, the data cannot come from user interactions. Instead, the data is contained in multiple files, one for each stream. Each node determines the streams it will initiate from the information in the *ne.dat* file.

*Q\_E\_Files* determines which files must be transmitted for which streams originating at the current node. The mapping of streams to files is done with a simple file (*filemapper.dat*), which contains the name of a stream and the name of the file containing it. Using the mapping file

allows for further flexibility in defining program operation. All of the specifications for each stream are found by starting with the *s\_obj* class for that stream and its associated functions. The *s2emap* class contains a list of the Ensembles containing the stream and references to their Ensemble objects. Given these references, the data from the Ensemble objects (*E\_obj*), at that node of G, the Stream object, *S\_obj*, for the stream and the 3 arrays *Required*, *Required\_S* and *Avail* can all be located. Some of the additional data maintained in *Gnodedata* is: mapping from virtual to real node, mapping of stream identifiers to and from Ensembles, and the Adjacency matrix for the node. In addition to the information discussed in 5.8.1.2, each *E\_obj* contains (among other data items): stream identifiers for streams in the Ensemble, the *PV* for each stream, and rules for merging streams (from *erules.dat*), as used in step 4.B of the algorithm (p. 81). The stream identifiers in *E\_obj* are used with the number of the Ensemble to index into the 3 arrays to determine the number of segments available and required at each step of the algorithm where this information is needed.

*Q\_E\_Files*, having determined which files are to be transmitted, starts a thread, *bfiller*, for each such file, to read the file and queue each segment of the file for further processing by *Send\_Ensembles*. Although these threads operate asynchronously, segment coordination within each Ensemble is managed by the Ensemble rules obtained in Stage 1 from *erules.dat* and the three arrays (*Required*, *Required\_S* and *Avail*) described in Stage 1. When *Q\_E\_Files* has started all the required threads, it terminates, leaving the threads to complete their operations.

When a segment of a stream is to be transmitted, information such as stream ID, and *Pathnode Vector* are marshaled to concatenate them with the stream segment. Using the stream ID, *bfiller*

determines which Ensembles use this stream, and then sets the *Avail* array to indicate **it is**

#### 5.9.1.4. *Out\_Qmgr*

*Out\_Qmgr* has the responsibility of allocating an output buffer, inserting the data in the buffer, enqueueing the buffer for *Sndthread* and performing the `Pthread_signal_condition` to awaken *Sndthread*. When it receives the signal from the `add_to_bufflist` method in *S\_obj*, *Out\_Qmgr*, exits its conditional wait, and begins performing `S_obj.get_Eseg` operations, to retrieve the segments for the current Ensemble/Stream combination. When `S_obj.get_Eseg` recognizes that it has retrieved a full Ensemble, it determines if those segments must be re-used for any other Ensemble. If not, the segments are purged from the output queue, using the *S\_obj* methods: `E_garbage_collection`, `expunge` and `adjust`. The first of these keeps track of the starting and ending segments (within the queue) of an Ensemble. When any sequence of segments is no longer required for a stream, those segments are deleted by a call to `expunge`. `Adjust` then modifies the indices of the arrays, *Avail* or *Avail\_S*. In either case, *Out\_Qmgr* is notified that there are no segments remaining to process now. The *Out\_Qmgr* then signals the *Sndthread* program to send each packet to its destination.

#### 5.9.1.5. *Sndthread*

*Sndthread* has the responsibility for handling all outgoing communications operations with each remote node, including error processing and establishing new connections to other nodes. As part of its operation, *Sndthread* must be able to determine if a connection already exists to a node, so it can re-use the socket. If a connection does not exist yet, *Sndthread* initiates a connection, asynchronously with other program operations, then stores the state of that connection and all other connection-relevant information, including the socket identifier, in the *Gnodedata* for that node. If the connection does exist, *Sndthread* retrieves all the socket

information from *Gnodedata* and initiates the data transfer then appends the end marker to each outgoing buffer.

The current implementation has only one thread for sending data, regardless of the number of streams and adjacent nodes.

## 5.9.2. Receiving data

Receiving data is performed by independent threads, each listening for data on its own TCPIP socket. The sockets (and their corresponding receive threads) are created when the primary socket does a TCPIP `accept` on a connection request from another node.

### 5.9.2.1. *Rcvthread*

*Rcvthread* is responsible for all incoming communications operations. When data is received, *Rcvthread* calls *In\_Qmgr*, which, by nature of a call acts as part of the calling thread, thus providing thread-safety. *In\_Qmgr* and *Q\_E\_Net* are separate threads, forming a traditional Producer-Consumer pair, with the former being the Producer. As mentioned in 5.2, each received TCP buffer might contain more than one stream segment. *Rcvthread* scans the full receive buffer for occurrences of the pseudo line-end character and strips out the preceding segment, presents it to *In\_Qmgr*, repeating this process until the entire receive buffer has been handled.

### 5.9.2.2. *In\_Qmgr*

The *In\_Qmgr* program unmarshals the fields of incoming data, examines the list of related Ensembles, inserts the Ensemble names in a queue of Ensembles ready to be processed, and puts

the segment on the incoming data queue. Again, a list of indices is kept to manage the locations of streams within the data queue. This list of indices is also used to remove all the segments of an Ensemble when the Ensemble has been completely processed. This garbage-collection operation is initiated, in the same manner in section 5.9.1, by the Stream Object, *S\_obj*, whenever an **S\_obj.get\_Eseg** operation determines from the indices and the three arrays, that the Ensemble processing will be completed (from the Ensemble management perspective) after the current retrieval completes. Any additional processing done by the application is independent of the segment queues. When all the segments for one stream that are required for an Ensemble have arrived, as determined by comparing *Avail(E,S)* with *Required(E,S)*, (where *E* is the Ensemble number and *S* is the stream number). *In\_Qmgr* uses the *S\_obj* method **add\_to\_bufflist**, to enqueue the packet on the queue for incoming data. **S\_obj.add\_to\_bufflist** signals either the *Win\_dispatcher* thread to display the data or the *Send\_ensembles* thread to requeue the data as an output segment. These two threads prepare the data to be either displayed or forwarded. When an Ensemble is complete (all segments are available), the ECP must decide what to do with it. If the data is to be displayed, the segments of the stream are dequeued and displayed, in order by stream within Ensemble in the proper Ensemble window. If the data is to be transmitted, the stream segments are forwarded, again in order by stream within Ensemble, to the next node in the Pathnode Vector, *PV*.

#### 5.9.2.3. *Q\_E\_Net*

*Q\_E\_Net* manages the queue of buffers for data coming from the network. It is signaled from *In\_Qmgr*, which has, at that point, already enqueued one or more buffers of data on the incoming data queue. *Q\_E\_Net* must extract this data and requeue it on the stream-specific queue to which it belongs. This Producer-Consumer arrangement obviates the need for the I/O

management programs (*Rcvthread* and *In\_Qmgr*) to pause I/O while unmarshaling the arguments of the I/O packet to process them. Since *Q\_E\_Net* is a separate thread from the I/O managers, it can afford to take the longer time for handling the meaning of the data. *Q\_E\_Net* then determines whether to signal *win\_dispatcher* to display the data or (in the case of an ISS) to signal the *Send\_Ensembles* program to re-send the data to the next node in the path for this stream.

### 5.9.3. Multi-casting in the ECP

One immediate advantage of the Distribution Vector/Pathnode Vector mechanism is that it reduces the number of re-broadcasts of data. Only those nodes which have more than one adjacent node in the list need to re-send the data, and then, only to those adjacent nodes which require the data. There is never a need for a node to ignore data, nor for the network to be managing broadcast data that is not relevant. Additionally, in the absence of graph re-configuration, complex multicast routing does not have to be computed at each re-broadcaster. The *PV/DV* mechanism provides a direct relationship between a stream segment and the nodes to which it must be sent. Even in the event of re-configuration, the routing computation would still be a simple AND operation. If dynamic re-configuration were implemented, the vectors would only have to be modified once during Stage 2 for all such changes. The underlying TCPIP protocol manages the actual routing to the graph-node specified in the TCPIP send. Another benefit of the PV and DV mechanism is that each segment contains complete information about its origin and destination, as well as every node to which must be sent. The program as written is designed to allow inclusion of a capability for any node to update other nodes regarding new Ensembles and streams to add to existing Ensembles.

## 5.10. Data structures Classes & Methods

In this section, we discuss some of the key data structures and C++ class methods defined in the program.

### 5.10.1. The Graph Node Descriptor (*Gnodedata*)

The collection of virtual network nodes that are members of the conference are described as a graph. During Stage 1, the graph is constructed as a LEDA Parameterized Directed Graph. Each node of the graph contains a data structure (a C++ class) named *Gnodedata* (Figure 9). Each *Gnodedata* structure contains a list (*Ensemble\_list*) of Ensemble Object structures (*E\_obj*), connection state, the handle of the socket being used (to a connected remote site), the remote port number and node type. Node types are: Source, Destination, Intermediate Service Station (ISS) or combined Source and Destination.

The *Gnodedata* structure is a C++ class with public and private functions (methods) to manipulate the various elements of the structure.

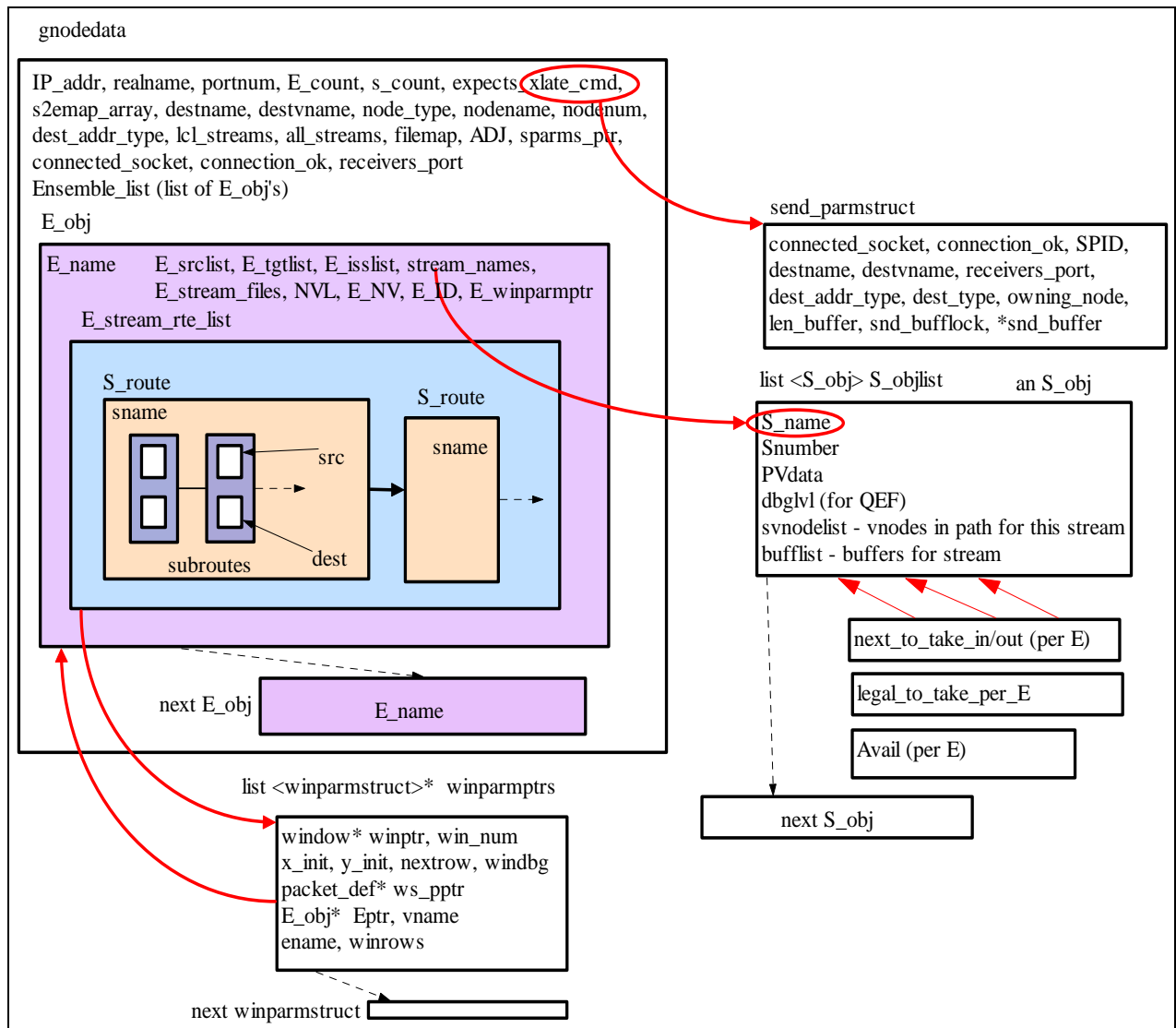


Figure 9. The Graphnode & E\_obj Descriptor Structures

### 5.10.2. The Ensemble Object (*E\_obj*)

Every Ensemble, whether managed at the current node or not, is represented, in the Gnode data structure with an *E\_obj* structure (see Figure 9), which is embedded in a C++ class with its own methods for managing the elements of the structure. Each *E\_obj* contains a list of all streams used in that Ensemble and a list (*E\_stream\_rte\_list*) of paths (*S\_route*'s) that each stream may traverse. Each route contains a Path Object class structure (*pathobj*) that contains a list of the

specific nodes traversed by the stream ordered as segment source and destination pairs. Each *pathobj* also contains the Pathnode Vector that is a string of bits representing the nodes to be traversed by the stream. Each *E\_obj* is cross-linked to every graphnode object (*Gnodedata*) that manages the Ensemble at any point in its travels.

### 5.10.3. The Stream Object (*S\_obj*)

The stream class object, *S\_obj*, contains all data relevant to each stream, as well as several C++ methods for manipulating the incoming and outgoing queues containing the stream. These methods include: adding to the queue (**add\_to\_bufflist**), retrieving without dequeuing (**get\_Eseg**), and dequeuing (**E\_garbage\_collection**, **expunge** and **adjust**). These operations control and are controlled by, the arrays: *Required*, *Required\_S* and *Avail*. All stream objects are in a linked list, *S\_objlist*, and are located by the name of the stream, as defined in the *ne.dat* file.

### 5.10.4. The Stream Object List (*S\_objlist*)

The Stream Object List maintains a list of buffers for the stream, a quick reference list of all nodes in the path for the stream, and a copy of the Path Vector for the stream, for validation.

### 5.10.5. The Stream Route Object (*S\_route*)

Each Ensemble Object contains a list of *S\_route*'s, one for each stream in the Ensemble. The *S\_route* is used to quickly determine which nodes are in the path of a given stream for its source to its destination. This information is computed in Stage-1 when the graph is created. Each *S\_route* contains a list of all nodes in the route for the stream as source-destination pairs. This maintains the order of nodes in the path traversal, for computing the Pathnode Vector (*PV*).

### 5.10.6. The Path List

Given the Directed Graph, G, of the nodes in Figure 6, the Adjacency Matrix for Directed graph G (from-nodes at the left) we can show an example of the run time computation of a *Path List*.

First, we develop the Adjacency Matrix for the graph. The result is Figure 10.

	SO1	SO2	SO3	SO4	SO5	D1	D2	D3	ISS1	ISS2	ISS3	ISS4
SO1	0	0	0	0	0	0	0	0	1	0	0	0
SO2	0	0	0	0	0	0	0	0	0	1	0	0
SO3	0	0	0	0	0	0	0	0	0	0	1	0
SO4	0	0	0	0	0	0	0	0	0	0	0	1
SO5	0	0	0	0	0	0	0	0	0	0	0	1
D1	0	0	0	0	0	0	0	0	0	0	0	0
D2	0	0	0	0	0	0	0	0	0	0	0	0
D3	0	0	0	0	0	0	0	0	0	0	0	0
ISS1	0	0	0	0	0	0	0	0	0	0	0	1
ISS2	0	0	0	0	0	0	0	0	0	0	0	1
ISS3	0	0	0	0	0	1	1	1	0	0	0	0
ISS4	0	0	0	0	0	0	0	1	0	0	1	0

Figure 10. Adjacency Matrix for the directed graph, G.

Each row of Figure 10 is an Adjacency Vector (AV), describing the nodes adjacent to the node whose name is in the heading of the row.

We now use the graph of Figure 6 to determine the routing for the stream from SO3 to D1 and D3, and to build the DV to control that routing through the network. Using the path search program, we find that the paths for the stream are:

1. SO3, ISS3, D1
2. SO3, ISS3, D3

Once these paths have been found, they are merged to form a single path-list with no duplicate entries.

The Path List for S3 is: SO3, D1, D3, ISS3

which has the same ordering of nodes as used to compute the adjacency matrix (the order of the headings in Figure 10). The same operations are performed for all streams.

### 5.10.7. The Pathnode Vector (*PV*)

The Pathnode Vector (*PV*), is a string of bits representing all the nodes in the graph, with a "1" for each node in the path-list and a "0" for those nodes, SO1, SO2, SO4, SO5 D2, ISS1, ISS2 and ISS4 that do not occur in the path-list. Using the *Path List* computed in 5.10.6, we get

$$PV = \mathbf{001001010011}$$

as the Pathnode Vector representing the nodes:

$$\overline{SO1}, \overline{SO2}, SO3, \overline{SO4}, \overline{SO5}, D1, \overline{D2}, D3, \overline{ISS1}, \overline{ISS2}, ISS3, \overline{ISS4}$$

where an overbar indicates a node is NOT in the path-list. The *PV* is used to determine whether the current segment is to be routed to other nodes or not.

### 5.10.8. Distribution Vector (*DV*)

A *DV* is an array of bits, 1 per graph node, where a "1" represents a node in the path for the current stream, from the current node to the next destination (one graph-edge distant).

To determine the routing from the current node, SO3, to the next node(s) in the graph, we perform a bitwise AND of the *PV* computed in the previous section (**001001010011**), with the *AV* for SO3 (row 3 of Figure 10, **000000000010**) giving the *DV* (**000000000010**) representing

ISS3 as the next destination for this element of the stream. At ISS3, the *AV* is **000001110000**. We perform the logical AND of this *AV* with the original *PV* (**001001010011**) to get the *DV* **000001010000** which corresponds to the nodes D1 and D3. These are thus the next (and final) destinations for the element of the stream from ISS3.

At D1 and D3, the element is recognized as being destined for those nodes (the resulting AND operation will be all zeros), and the element will be displayed, with its corresponding elements from SO1 and SO2, when they arrive, in the window created for Ensemble E0.

#### 5.10.9. The Send/Receive Packet (*packet\_def*)

```
typedef struct
{
    string      p_type;           // 0=intersite command, 1=data for Ensemble
    string      bufflen_str;     // databuffer byte count
    string      PVdata;         // target & distribution info
    string      Ensemble_names;  // all ensembles using this stream
    string      StreamID;       // name of the stream
    string      databuffer;     // actual stream data
} packet_def;
```

This structure is the actual data packet used to send and receive the segments of each stream of data or inter-node command. The packet terminator, used for testing with text strings, is not shown.

## 6. Inputs and Outputs

### 6.1. *The Node and Ensemble Definition File (ne.dat)*

The *ne.dat* file contains:

1. A list of virtual node names for each node of the graph, with their corresponding real names. The real names are the machine host names of the machines for each node.
2. A list of the nodes and edges of the graph (using the pseudonyms)
3. A list of Ensemble names, the streams they contain and their sources and final destinations (both by pseudonym).

The use of virtual nodes allows for greater flexibility in assigning machines to positions within the conference graph. This means a simple name change on one line can put a graph node on a different computer. The *node2ip.dat* file is similar to the standard ETC/HOSTS file used by TCPIP to identify node names not serviced by a nameserver with their TCPIP dotted-decimal addresses.

## 6.2. The host locator file (*node2ip.dat*)

Many operating systems maintain a file in the ETC directory called HOSTS. This file is a lookup file for matching a domain name to an IP address for use by all programs running on a system. The HOSTS file is used when a Distributed Name Server (DNS does not have the name). The *node2ip.dat* file is similar in function, but is only for use by the ECP.

## 6.3. The Ensemble Rule Descriptor File (*erules.dat*)

In this implementation, the expression for the elements of an Ensemble is input from a rule definition file (*erules.dat*). The file contains statements in the following format:

```
Ename (sname, quant) (sname, quant) ... ;
```

where *Ename* is the name of the Ensemble being described, *sname* is the name of the stream and *quant* is the number of segments of that stream required to complete the Ensemble before it can be transmitted or presented to the application. An example of a conference with 3 Ensembles, consisting of various combinations of 4 different data streams, might be represented in *erules.dat* as:

```
E1 (s1,2) (s2,1) (s3,4);  
E2 (s2,1) (s3,2);  
E3 (s1,1) (s2,2) (s3,2) (s4,3);
```

Notice that streams *s1*, *s2* and *s3* are all used in more than one Ensemble and that they do NOT require the same number of segments per Ensemble. Such requirements would come from the

nature of the individual streams, their inter-relationships and the purpose of the conference and meaning of such Ensembles.

The file used for the test in this thesis, based on the network in Figure 6 on page 27 contains the following rules:

E1 (s1,3) (s2,2) (s3,3);  
E2 (s4,2) (s5,2);

The *erules.dat* file is read at initialization of stage 1 and the rules are stored in the previously described arrays: *Required*, and *Required\_S*.

#### **6.4. Input Data (s1.dat through s5.dat)**

As previously discussed, the input for the program is simulated with files. There are 5 of these files, *s1.dat* through *s5.dat*. Each file consists of records in the following format:

streamname "record text"

where the "record text" includes the number of the record within the file.

Thus a typical file, *s3.dat*, for example would have records that look like the following 3 records:

s3 record 1  
s3 record 2  
s3 record 3

If we use this method of identification for every record within each stream, we will be able to track the data within the output for each Ensemble, as well as in the timing output file.

## **6.5. Output Data**

As each window receives a segment (representing a record of data for the application program), the segment is written to a window capture file. The data in this file is not modified in any way, so it is readily apparent when an Ensemble has been completed. Following is a copy of this file for Ensemble E1, (see Figure 6), recorded at node D1, for the case when all files have 30 records.

## **6.6. Output of E1 at nodes D1 and D3**

The following data in Figure 11 on page 80 was recorded in the output window at nodes D1 and D3 for Ensemble E1, and was identical for both nodes.

s1 record 1  
s1 record 2  
s1 record 3  
s2 record 1  
s2 record 2  
s3 record 1  
s3 record 2  
s3 record 3

=====

s1 record 4  
s1 record 5  
s1 record 6  
s2 record 3  
s2 record 4  
s3 record 4  
s3 record 5  
s3 record 6

=====

s1 record 7  
s1 record 8  
s1 record 9  
s2 record 5  
s2 record 6  
s3 record 7  
s3 record 8  
s3 record 9

=====

s1 record 10  
s1 record 11  
s1 record 12  
s2 record 7  
s2 record 8  
s3 record 10  
s3 record 11  
s3 record 12

=====

s1 record 13  
s1 record 14  
s1 record 15  
s2 record 9  
s2 record 10  
s3 record 13  
s3 record 14  
s3 record 15

=====

s1 record 16  
s1 record 17  
s1 record 18  
s2 record 11  
s2 record 12  
s3 record 16  
s3 record 17  
s3 record 18

=====  
s1 record 19  
s1 record 20  
s1 record 21  
s2 record 13  
s2 record 14  
s3 record 19  
s3 record 20  
s3 record 21

=====

s1 record 22  
s1 record 23  
s1 record 24  
s2 record 15  
s2 record 16  
s3 record 22  
s3 record 23  
s3 record 24

=====

s1 record 25  
s1 record 26  
s1 record 27  
s2 record 17  
s2 record 18  
s3 record 25  
s3 record 26  
s3 record 27

=====

s1 record 28  
s1 record 29  
s1 record 30  
s2 record 19  
s2 record 20  
s3 record 28  
s3 record 29  
s3 record 30

Figure 11. Output Recorded for E1 at both D1 and D3

For clarity, before including the file in this thesis, the Ensembles in the file were demarcated with a row of "==" signs. For this Ensemble, the rule in *erules.dat* was:

E1 (s1, 3) (s2, 2) (s3, 3);

Note that there are 10 records from Stream 2 that do not appear at the end of the data. When all nodes finished their transmissions, these 10 records had no Ensemble data to combine with and were left in the queue. This situation is caused by the nature of the test data, rather than as a problem in the program. The files used to test the program all contain the same number of records for this test set (30 records), and the Ensemble rules for the conference do not distinguish any difference among the files, other than that they represent different streams of data. This independence allows for rapidly modifying the test cases by simply changing the entries in *erules.dat*.

## 7. The Coordination Algorithm

1. Create the graph and an array representing every node of the graph.
2. From the graph, compute the Adjacency Vector ( $AV$ ) for the current node.
3. If this is a destination node, create a window for each Ensemble to be "played" at this node. This Window is an example of an application that would present the data to the user.
4. For each Ensemble originating at this node:
  - A. For each stream:
    - 1) Given the starting and ending locations for the stream, use the graph to determine the paths to all the destinations for the stream. This is the Path List
    - 2) Create the Pathnode Vector ( $PV$ ) for the stream. The  $PV$  is a string of bits, 1 per graph node. If the node is in the Path List set the bit for that node to "1". Else, set the bit to "0".
    - 3) Compute the Distribution Vector ( $DV$ ) for the stream.
  - B. For each segment in each stream of this Ensemble originating at this node:
    - 1) Determine if all required Ensemble elements are available.
    - 2) Use the rules for coordinating the streams of the Ensemble to create a list of elements to be forwarded at this time.
      - a. Obtain the stream elements of the Ensemble from the queue
      - b. Attach the  $PV$  for the stream to each segment of the stream

- c. Marshal the content of a packet
  - d. Send the packet of this datastream to those nodes represented by 1's in the *DV*.
5. For each Ensemble passing through this node (not originating at this node):
- A. Determine if all required Ensemble elements are available.
  - B. Use the rules for coordinating the streams of the Ensemble to create a list of elements to be forwarded at this time.
  - C. Extract the *PV* from the stream element. Compute the *DV*. Use the *DV* to determine the next nodes to receive this stream element. To do this:
    - 1) Perform the logical AND of the bits of *PV* with the bits in *AV* for the current node, to obtain a new *DV*.
    - 2) Send the elements of this datastream to those nodes represented by 1's in the *DV*.
  - D. If this is a Destination Node, display the data

## **8. Testing, Analysis and Results**

### **8.1. Testing**

#### 8.1.1. Initialization

Testing involved the following steps, based on the graph in Figure 6:

1. Establish Virtual Node Definitions
  - a SO1 (Win/ME): (dynamically assigned by Internet Service Provider-this was the master node)
  - b SO2 (UNIX): sapphire.cs.binghamton.edu
  - c SO3 (UNIX): pearl.cs.binghamton.edu
  - d SO4 (WinNT): poohbear.hundredacres.binghamton.edu
  - e SO5 (WinNT): rabbit.hundredacres.binghamton.edu
  - f ISS1 (UNIX): bingsuns.binghamton.edu
  - g ISS2 (UNIX): onyx.cs.binghamton.edu
  - h ISS3: (UNIX) coral.cs.binghamton.edu
  - i ISS4 (UNIX): ruby.cs.binghamton.edu
  - j D1 (Win/ME): (dynamically assigned by Internet Service Provider)
  - k D2 (UNIX): sapphire.cs.binghamton.edu

## 1 D3 (UNIX): agate.cs.binghamton.edu

These definitions are inserted in the *ne.dat* file in the form:

virtual node systemname

for example: SO3 pearl.cs.binghamton.edu

The system names are then inserted in the *node2ip.dat* file in the form:

systemname IP address port number

for example: pearl.cs.binghamton.edu 128.226.140.163 49999

## 2. File Management

- a. Create the data files (*s1.dat*, etc.) required at each source node
- b. Copy the *node2ip.dat* file to every node
- c. Copy the *ne.dat* file to every node
- d. Copy the *erules.dat* file to every node

## 3. Start the program

- a. Open an Xterm window for each Unix machine, using Exceed.
- b. From the master node (Win ME), open a VNC window for each Winxx machine other than the master node. Note: any node could be the master node, since this is a parameter to the program. The only purpose of having a master node is to allow time for all other nodes to be ready before beginning transmissions. This avoids the requirement for adding code that would accomplish the same task.
- c. Start the ECP on all machines, except the master.
- d. Select the desired settings on each system's startup menu.

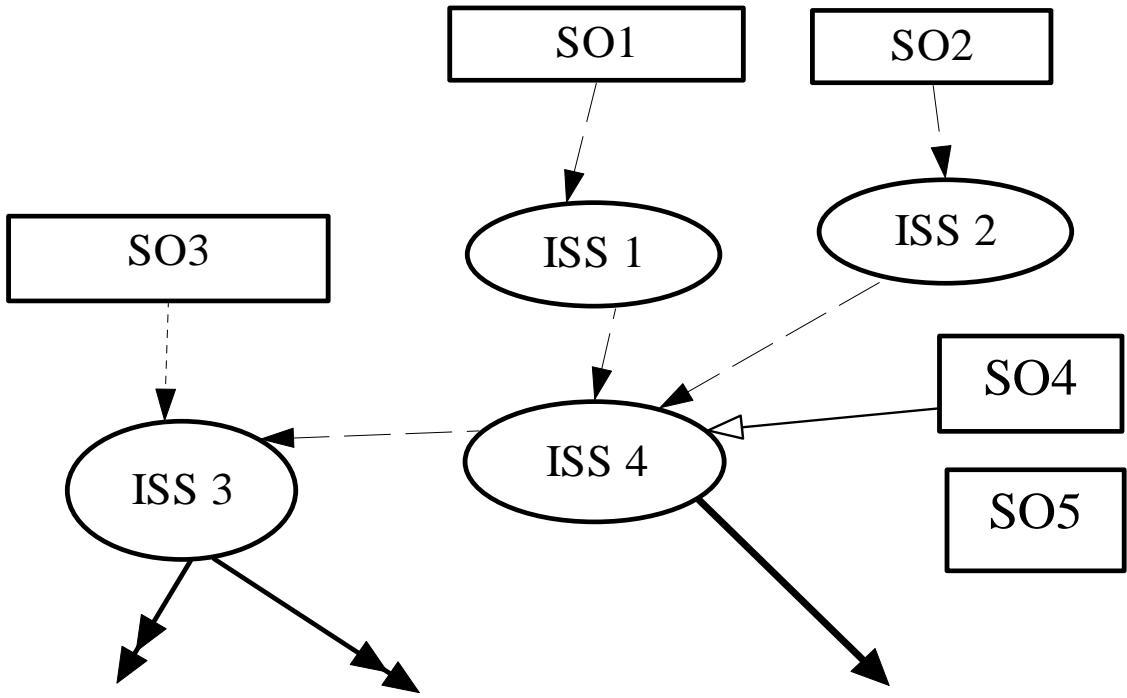
- e. After all nodes have actually started, start the ECP on the master node, select its debug settings and proceed with running the program.



#### 4. Operation of the Conference


- a. Each node inputs the system graph and builds its control structures as previously described, then waits for a signal from the master system that all nodes are available
- b. The master system communicates with all systems in the graph, including itself if it is a source, that all systems are ready
- c. All sources begin transmitting data according to the Ensemble rules, and the ISS's forward the data to the destinations as determined by the DV and PV for the data.
- d. When all transmissions have completed, withdraw each machine from the conference by selecting this option in the window opened by each node, on the master machine's monitor.

#### 8.1.2. Running the Program

In this section we revisit Figure 6 and describe in detail what happens during the transmission of data streams from SO1, SO2, SO3, SO4 and SO5. After initialization is completed, these nodes, having determined, from the analysis of the graph and the node structures built from the *ne.dat* file, that they are Source nodes (their in-degree is zero and their out-degree is non-zero), they open their Stream files and prepare to send them according to the Ensemble rules and the *PV* and *AV* information. SO1 will open *s1.dat* and use the *DV* computed in section 5.10.6 to send the data to ISS1. SO1 will examine the arrays, *Required* and *Avail*, and will find that 3 records of the file must be read and sent to complete an Ensemble at node SO1. Reading of the records from the files is done with asynchronous threads, one per file, thus allowing for multiple files at the same node.



Legend: Ensemble 1  

Ensemble 2 

When the records from SO1 reach ISS1, they are queued for re-transmission. When examination of the arrays, *Required* and *Avail* at ISS1 shows that no further data is required to complete an Ensemble at this node, the data is again transmitted, using the *PV* and *AV* to compute the *DV* to determine the next node, which in this case will be ISS4. The same actions occur at SO2, with data being output to ISS2. ISS2 will also be sent to ISS4. The data from SO3 will be sent to ISS3.

When the data arrives at an ISS or a Destination node, the value in *Avail* is updated so it can be used as a comparison against *Required* to again determine if an Ensemble is complete.

From here on, the actions are more complex, as they require the ISS's to track the first record of each Ensemble, their current position in each stream, the number of records available in each stream and the number of records already transmitted in each stream. This is done with a set of arrays (*start*, *next\_to\_take* and *left\_to\_take*) that contain pointers to the buffers containing the segments of data. *Start* indicates the first segment of a stream for the Ensemble. *Next\_to\_take* indicates the position in the stream for the next segment to be transmitted, and *left\_to\_take* contains a count of how many more segments must be sent before the stream transmission is complete. Separate arrays are maintained for each stream in each Ensemble Set. This allows a stream to be used in multiple Ensemble Sets. Once *left\_to\_take* reaches zero, if there are no other Ensembles requiring this stream, the segments are removed from the queue and the arrays are updated for the next group of segments. If the segments are to be re-used for another Ensemble, they are not purged.

Using these arrays and counters, ISS3 and ISS4 determine the segments to be transmitted and transmit them, again using the *DV* information computed the same way as noted for SO1. Again referring to the duplicate of Figure 6, the data from ISS4 (streams S1 and S2) are sent to ISS3 and D3, and the data from Stream SO3 is sent from ISS3 to D1 with the data from ISS4 when it arrives and directly to D3, to be merged there with the data from ISS4.

When the data arrives at a Destination node, such as D1 or D3, the arrays, *Required* and *Avail* are again used to determine if an Ensemble is complete and ready for display (or to be sent to an application). When the data is no longer needed for another Ensemble, the data is purged from the input queue.

### 8.1.3. Measurements

Data segments are time-stamped with elapsed clock-ticks (*clock\_t*) converted to milliseconds.

$$\text{time} = \text{clock}() / (\text{CLOCKS\_PER\_SEC} / (\text{clock\_t})1000)$$

*CLOCKS\_PER\_SEC* and *clock\_t* are macro variables provided by the C++ library. The former specifies the number of CPU clock ticks there are per second, and the latter is a *type\_cast* operator for the divisor.

The initial time stamp is applied by the program at either of two locations, depending on whether the data is being transmitted or received. For input processing, the data is time stamped when it is inserted by the *In\_Qmgr* into the incoming-data queue. For output processing, it is time stamped when it is first inserted into the output-queue by *Send\_Ensembles* (for a Source node) or *win\_dispatcher* (for a Destination node).

## 8.2. Analyzing the Ensemble timing data

From the timestamps corresponding to the Ensemble data in 6.5, we get the file below. Times are in elapsed milliseconds since the program started. From the data below, we see that the first Ensemble for E1 began to arrive at 106830, but was not displayed until 135290 msec later. In other words, the data for this Ensemble took  $122870 - 106830 = 16040$  msec to arrive, but only  $153570 - 153290 = 280$  msec to display. The second Ensemble began arriving at the same time as group 1, but streams s2 and s3 arrived significantly later, so this Ensemble was not displayed until  $194540 - 106830 = 87710$  msec after program start, or  $194540 - 153290 = 41250$  msec after Ensemble 1 completed. From the greatly varying arrival and display times, we can see that without any imposition of QoS requirements on the network, the delays (affecting the value of  $\delta$  discussed in section 3.4) are less affected by the fixed amount of processing time, as by network traffic and other work being done on the test systems. These systems are in laboratories open to the entire Computer Science student population, the machines support multiple simultaneous users and processing loads vary significantly from machine to machine and from one time of day to another.

In Figure 12, the fields are:

*Ensemble name, stream name, segment handler, time into queue, time removed from queue*

The segment handler, in this case, is the **Window\_Draw** function, simulating the action of the multimedia application.

E1/s1/WD	/106830/153290/
E1/s2/WD	/112700/153400/
E1/s3/WD	/122870/153570/
E1/s1/WD	/106830/153290/
E1/s2/WD	/168120/194380/
E1/s3/WD	/184820/194540/
E1/s1/WD	/106830/153290/
E1/s2/WD	/197840/201740/
E1/s3/WD	/199320/201960/
E1/s1/WD	/106830/153290/
E1/s2/WD	/204050/208060/
E1/s3/WD	/205360/208280/
E1/s1/WD	/106830/153290/
E1/s2/WD	/211900/217450/
E1/s3/WD	/213990/217610/
E1/s1/WD	/106830/153290/
E1/s2/WD	/221570/224860/
E1/s3/WD	/222560/225030/
E1/s1/WD	/106830/153290/
E1/s2/WD	/227220/230740/
E1/s3/WD	/228430/230900/

Figure 12. Recorded Timing Data for E1

### 8.3. Results

We can see that the resultant file (see section 6.6) does indeed contain 3 records of stream 1, followed by 2 records of stream 2 and 3 records of stream 3, in repeating sequence, until the data is exhausted. We can see from this data that the 3 Ordering Properties have been maintained as follows:

1. Single source: the orders of records in each stream (s1, s2, s3) are observed to be in proper sequence across all Ensembles in the test.

2. Multiple sources: the order of records from each source is observed to be constant across Ensembles.
3. Multiple destinations: the results observed at D1 were observed to be identical to the results obtained at D3.

In addition to the Ordering Properties, the first 3 Synchronization Types were maintained:

- I. Intra-object: Within a single file, this is guaranteed by TCPIP and it is evident from the fact that the text within each file is in proper order. Within a stream, we see, in 6.6, for any stream  $S_n$ , the records are in the same sequential order as described for their source (in section 6.4). Therefore, since the records are maintained in proper order, the application will be able to maintain the time relationships between records.
- II. Inter-object: We can see that the records for each stream received in E1 maintain the same sequential relationship with the records of the other streams, for every Ensemble.
- III. Inter-site: Since Type II Synchronization is maintained, and the Ensembles are identical at multiple sites (D1 and D3), I concluded that the Ensembles at each node have the same Synchronization attributes and Inter-site Synchronization was maintained.

Incoming data segments are individually enqueued for processing and are time-stamped as described above. When a data segment is removed from its queue, the identifying information for that segment and its timestamp data are saved to an output file, which is analyzed separately.

## 9. Conclusions

As seen in section 6.6, the identification of records proposed in section 6.4 did provide a useful means of record tracking, which enabled confirmation of the Ordering Properties we set out to establish.

The proposed method of approach is complete and the stated objectives have been met:

1. I determined a means for identifying the elements of the Ensemble such that the Ensemble can be formed even if the elements originate at physically different locations (see section 6.2).
2. I showed that we can dynamically specify inter-stream synchronization parameters, using the expression  $E_{M,N,\dots}(x,y,\dots)$  (see section 6.2).
3. I transmitted the Ensembles in proper order, with proper sequencing and correct coordination of their elements, to multiple recipients, using a simple Distribution Vector, thus maintaining all 3 Ordering Properties, across multiple collections of streams of data (see section 6.6), while allowing for the required Synchronization Types (I and II) as well as the desired Type III. We have not attempted to solve the problem of type IV (Global) Synchronization.
4. I merged elements from multiple sites (an Ensemble Set), into the Ensemble transmission (see section 6.6) and delivered them in the proper order.

5. I defined a set of rules which determine how each Source and ISS (Independent Synchronization Server) determines which streams to send, when to send them and in what directions to send them (section 5.10.6, 5.10.7).
6. I included a capability for the application to dynamically specify the values of the expression,  $E_{M, N, O}(x, y, z)$ , as discussed above, and hence, to control the Ensemble Coordination, by specifying the file containing these parameters (see section 6.2).

## 10. Future Work

There are many areas of follow-on work to consider, such as:

1. An interface for changes to the graph to allow:
  - a. Adding ISS nodes, Sources and Destinations.
  - b. Re-configuring the graph edges without changing node types.
  - c. Re-configuring the graph edges, thus changing node types.
  - d. Re-configuring nodes to act as other types.
2. More clearly separate the Application Programming Interface (API) from the underlying coordination code, at which time the simulated application should be removed.
3. Allow more dynamic specification of data sources.
4. Allow nodes to be dynamically added to or removed from the conference.
5. Test with non-textual data such as video, music and other such sources. This would be dependent on completion of the changes to the API.
6. Remove the requirement for a master node.
7. Explore the possibilities of providing Type IV (Global) Synchronization.
8. Extension of LEDA window support to the OS/2 environment.
9. Extension of Pthreads to OS/2.

## 11. Bibliography

- [Adl95] Richard M. Adler, "Distributed Coordination Models for Client/Server Computing," *Computer*, April 1995, p. 14.
- [Ahu92] Sudhir R. Ahuja and J. Robert Ensor, "Coordination and Control of Multimedia Conferencing," *IEEE Communications Magazine*, May 1992, pp38-43.
- [AY96] Ian F. Akyildiz and Wei Yen, "Multimedia Group Synchronization Protocols for Integrated Services Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 1, January, 1996, p. 162.
- [ABLM95] B. K. Aldred, G. W. Bonsall, H. S. Lambert, H. D. Mitchell, "An Architecture for Multimedia Communication and Real-Time Collaboration", *IBM Systems Journal*, Vol. 34, No. 3, 1995, p. 519.
- [AWG94] Zafar Ali, Miae Woo and Arif Ghafoor, "Distributed Synchronization Protocols for Multimedia Services on Internet," Proc. of the 1994 Int'l Conf. on Network Protocols, IEEE, 12/94, p. 206.
- [AH91] David P. Anderson and George Homsy, "A Continuous Media I/O Server and Its Synchronization Mechanism," *Computer*, October 1991, p. 51.
- [And93] David P. Anderson, "Metascheduling for Continuous Media," *ACM Transactions on Computer Systems*, Vol. 11, August 1993, p. 226.
- [BFC93] Anthony J. Ballardie, Paul Francis, and Jon Crowcroft, "Core Based Trees", *Computer Communications Review*, Vol 23, No. 4, October, 1993, p. 85.

- [BKTZ94] A. Banerjea, E. Knightly, F. Templin and H. Zhang; “Experiments with the tenet real-time protocol suite on the Sequoia 2000 wide area network”, *Proceedings of the second ACM international conference on Multimedia*, 1994, Pages 183 – 191.
- [BS96] Gerold Blakowski and Ralf Steinmetz, “A Media Synchronization Survey: Reference Model, Specification, and Case Studies”, *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 1, January, 1996, p. 5.
- [Bra93] Barbara Bracken, “General Multimedia Consultative Conferencing support in a Distributed Interactive Digital Multimedia system,” Master’s Thesis, Binghamton University, 1993.
- [But97] David Butenhof, “Programming with POSIX Threads,” Addison Wesley, 1997.
- [CFC\*96] Jean-Pierre Courtiat, Luiz Fernando Rust da Costa Carmo, and Roberto Cruz de Oliveira, “A General-Purpose Multimedia Synchronization Mechanism Based on Causal Relations”, *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 1, January, 1996, pp. 185-195.
- [DEF\*94] Steven Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu and Liming Wei, “An Architecture for Wide Area Multicast Routing”, *Computer Communication Review*, Vol. 24, No. 4, October, 1994, p. 126.
- [Dua95] Jose Duato, “A theory of deadlock-free adaptive multicast routing in wormhole networks.” *Transactions on Parallel and Distributed Systems*, Vol 6, Sept 1995, pp 976-987.
- [For01] Dennis J. Foreman, “A data routing and coordination infrastructure for use by distributed multimedia conferencing applications”, Nov 2001, accepted for publication, *IEEE Multimedia Magazine*.
- [FT01] Mark Furini and Donald Towsley, “Real-Time Traffic Transmissions over the Internet”, *IEEE Transactions on Multimedia*, Vol. 3, No. 1, March 2001.

- [GS91] Hector Garcia-Molina and Annemarie Spauster, "Ordered and Reliable Multicast Communication", *ACM Transactions on Computer Systems*, Vol. 6, No 3, August 1991, pp.242-271.
- [GSA01] Donna Ghosh, Venkatesh Sarangan and Raj Acharya, "Quality of Service Routing in IP networks", *IEEE Transactions on Multimedia*, Vol 3, No 2, June 2001.
- [Hai96] Michal Haindl, "A New Multimedia Synchronization Model", *IEEE Journal On Selected Areas in Communications*, Vol. 14, No. 1, January, 1996, p. 73.
- [IM91] Hiroshi Ishii, Naomi Miyake, "Toward an Open Shared Workspace: Computer and Video Fusion Approach of Teamworkstation", *Communications of the ACM*, Vol. 34, No. 12, December 1991, p. 37.
- [Joh] Ross Johnson, The Pthreads-Win32 Library and Source files, <http://sourceware.cygnum.com/threads-win32/>.
- [JSRM95] Paul W. Jardetzky, Cormac J. Sreenan, Roger M. Needham; "Storage and synchronization for distributed continuous media," *Multimedia Systems*; Vol. 3 No. 4, 1995; p. 151.
- [KO94] Lutz Kleinholtz and Martin Ohly, "Supporting Cooperative Medicine: The Bermed Project", *IEEE Multimedia*, Winter, 1994, p. 44.
- [KO94B] Lutz Kleinholz and Martin Ohly, "Multimedia Medical Conferencing: Design and Experience in the BERMED Project", *Multimedia Conferencing*, December 1994, pp.255-264.
- [LEDA1] LEDA, "A Library of Efficient Data Structures and Algorithms," [http://www.algorithmic-solutions.com/as\\_html/research/research\\_en\\_academy.html](http://www.algorithmic-solutions.com/as_html/research/research_en_academy.html).
- [LB98] Bil Lewis and Daniel Berg, "Multithreaded Programming with Pthreads," Prentice Hall, 1998

- [LMN94] Xiaola Lin, Philip McKinley, Lionel M. Ni, "Deadlock-Free Multicast Wormhole Routing in 2-D Mesh Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, August 1994, p. 793.
- [MOO87] Maekawa, Oldehoeft, Oldehoeft, *Operating Systems Advanced Concepts*, The Benjamin/Cummings Publishing Company, Inc., 1987.
- [MX\*94] Philip McKinley, Hong Xu, Abdol-Hossein Esfahanian, Lionel Ni, "Unicast-Based Multicast Communication in Wormhole-Routed Networks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 12, December, 1994, p. 1252.
- [Moy94] John Moy, "Multicast routing extensions for OSPF", *Communications of the ACM*, Vol 37, No 8, August 1994, p. 61.
- [PSK94] Sanjoy Paul, Krishna K. Sabnani and David M. Kristol, "Multicast Transport Protocols for High-Speed Networks", *Proc. of the 1994 International Conference on Network Protocols*, 1994, p 4.
- [PLL96] Maria Jose Perez-Luque and Thomas D. C. Little, "A Temporal Reference Framework for Multimedia Synchronization", *IEEE Journal on Selected Areas In Communication*, Vol. 14, No. 1, January, 1996, p. 36.
- [RR93] S. Ramanathan and P. Venkat Rangan, "Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems", *The Computer Journal*, Vol. 36, No. 1, 1993, p. 19.
- [Ran91] P. Venkat Rangan, "Software Architecture for Integration of Video Services in the Etherphone System," *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 9, December 1991, pp. 1395-1404.
- [Ran93V] P. Venkat Rangan, "Video conferencing, File Storage, and Management in Multimedia Computer Systems," *Computer Networks and ISDN Systems*, 25, 1993, p. 901.

- [Ran93T] P. Venkat Rangan, Srinivas Ramanathan, Harrick M. Vin, and Thomas Kaepfner, "Techniques for Multimedia Synchronization in Network File Systems", *Computer Communications*, Vol. 16, No. 3, March, 1993, p. 168.
- [RMC95] David F. Robinson, Philip K. McKinley, Betty H. C. Cheng, "Optimal multicast communication in wormhole-routed torus networks", *Transactions on Parallel and Distributed Systems*, October, 1995, Vol 6, p. 1029-42.
- [RH96] Kurt Rothermel and Tobias Helbig, "Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams", *IEEE Journal on Selected Areas in Communication*, Vol. 14, No. 1, January, 1996, p. 174.
- [Sch94] James A. Schnepf, "Building Future Medical Education Environments Over ATM Networks", *Communications of the ACM*, Vol. 38, No. 2, April, 1994, pp54-69.
- [Ste90] Ralf Steinmetz, "Synchronization Properties in Multimedia Systems", *IEEE Journal On Selected Areas in Communications*, Vol. 8, No. 3, April, 1990, page 401.
- [Ste95] Ralf Steinmetz, "Analyzing the Multimedia Operating System", *Multimedia*, Spring, 1995, Vol. 2, No. 1, p. 68.
- [SUN98] SunOS 5.7, man pages (threads) 25 Jun 1998.
- [VZSR91] Harrick M. Vin, Polle T Zellweger, Daniel C. Swinehart, and P. Venkat Rangan, "Multimedia Conferencing in the Etherphone Environment," *IEEE Computer*, Oct. 1991, pp 69-79.
- [YWS95] Philip S. Yu, Joel L. Wolf, Hadas Shachnai, "Design and analysis of a look-ahead scheduling scheme to support pause-resume for video-on-demand applications", *Multimedia Systems*, Vol. 3 No. 4, 1995, p. 137.

[ZZZ01]

Qian Zhang, Wenwu Zhu, and Ya-Qin Zhang, "Resource Allocation for Multimedia Streaming Over the Internet", *IEEE Transactions on Multimedia*, Vol. 3, No. 3, Sept 2001.